```
 ______   ___    ___
/\  _  \ /\_ \  /\_ \
\ \ \L\ \\//\ \ \//\ \      __     __   _ __   ___
 \ \  __ \ \ \ \  \ \ \   /'__`\ /'_ `\/\`'__\/ __`\
  \ \ \/\ \ \_\ \_ \_\ \_/\  __//\ \L\ \ \ \//\ \L\ \
   \ \_\ \_\/\____\/\____\ \____\ \____ \ \_\\ \____/
    \/_/\/_/\/____/\/____/\/____/\/___L\ \/_/ \/___/
                                   /\____/
                                   \_/__/     Version 4.0.3


              A game programming library.

           By Shawn Hargreaves, Apr 19, 2003.

              See the AUTHORS file for a
             complete list of contributors.
```

#include <std_disclaimer.h>

"I do not accept responsibility for any effects, adverse or otherwise, that this code may have on you, your computer, your sanity, your dog, and anything else that you can think of. Use it at your own risk."

1 Using Allegro

See readme.txt for a general introduction, copyright details, and information about how to install Allegro and link your program with it.

1.1 install_allegro

`int install_allegro(int system_id, int *errno_ptr, int (*atexit_ptr)());`

Initialises the Allegro library. You must call either this or allegro_init() before doing anything other than using the Unicode routines. If you want to use a text mode other than UTF-8, you can set it with set_uformat() before you call this. The available system ID codes will vary from one platform to another, but you will almost always want to pass SYSTEM_AUTODETECT. Alternatively, SYSTEM_NONE installs a stripped down version of Allegro that won't even try to touch your hardware or do anything platform specific: this can be useful for situations where you only want to manipulate memory bitmaps, such as the text mode datafile tools or the Windows GDI interfacing functions. The errno_ptr and atexit_ptr parameters should point to the errno variable and atexit function from your libc: these are required because when Allegro is linked as a DLL, it doesn't have direct access to your local libc data. atexit_ptr may be NULL, in which case it is your responsibility to call allegro_exit manually. Currently this function always returns zero. If no system driver can be used, the program will abort.

See also:

See Section 1.2 [allegro_init], page 1.

See Section 1.3 [allegro_exit], page 1.

1.2 allegro_init

`int allegro_init();`

Macro which initialises the Allegro library. This is the same thing as calling install_allegro(SYSTEM_AUTODETECT, &errno, atexit).

See also:

See Section 1.1 [install_allegro], page 1.

See Section 1.3 [allegro_exit], page 1.

1.3 allegro_exit

`void allegro_exit();`

Closes down the Allegro system. This includes returning the system to text mode and removing whatever mouse, keyboard, and timer routines have been installed. You don't normally need to bother making an explicit call to this function, because allegro_init() installs it as an atexit() routine so it will be called automatically when your program exits.

See also:

See Section 1.1 [install_allegro], page 1.

See Section 1.2 [allegro_init], page 1.

1.4 END_OF_MAIN

`Macro END_OF_MAIN()`

In order to maintain cross-platform compatibility, you have to put this macro at the very end of your main function. This macro uses some 'magic' to mangle your main procedure on platforms that need it like Windows or Linux. On the other platforms this macro compiles to nothing, so you don't have to #ifdef around it. Example:

```
int main(void)
{
   allegro_init();
   /* more stuff goes here */
   ...
   return 0;
}
END_OF_MAIN()
```

See also:

See Chapter 35 [Windows], page 234.

See Chapter 36 [Unix], page 242.

See Chapter 39 [Differences], page 248.

1.5 allegro_id

`extern char allegro_id[];`

Text string containing a date and version number for the library, in case you want to display these somewhere.

1.6 allegro_error

`extern char allegro_error[ALLEGRO_ERROR_SIZE];`

Text string used by set_gfx_mode() and install_sound() to report error messages. If they fail and you want to tell the user why, this is the place to look for a description of the problem.

See also:

See Section 8.6 [set_gfx_mode], page 70.

See Section 23.5 [install_sound], page 160.

1.7 os_type

`extern int os_type;`

Set by allegro_init() to one of the values:

```
OSTYPE_UNKNOWN    - unknown, or regular MSDOS
OSTYPE_WIN3       - Windows 3.1 or earlier
OSTYPE_WIN95      - Windows 95
OSTYPE_WIN98      - Windows 98
OSTYPE_WINME      - Windows ME
OSTYPE_WINNT      - Windows NT
OSTYPE_WIN2000    - Windows 2000
OSTYPE_WINXP      - Windows XP
OSTYPE_OS2        - OS/2
OSTYPE_WARP       - OS/2 Warp 3
OSTYPE_DOSEMU     - Linux DOSEMU
OSTYPE_OPENDOS    - Caldera OpenDOS
OSTYPE_LINUX      - Linux
OSTYPE_SUNOS      - SunOS/Solaris
OSTYPE_FREEBSD    - FreeBSD
OSTYPE_NETBSD     - NetBSD
OSTYPE_IRIX       - IRIX
OSTYPE_QNX        - QNX
OSTYPE_UNIX       - Unknown Unix variant
OSTYPE_BEOS       - BeOS
OSTYPE_MACOS      - MacOS
```

See also:

See Section 1.2 [allegro_init], page 1.

See Section 1.8 [os_version], page 3.

See Section 1.9 [os_multitasking], page 4.

1.8 os_version

`extern int os_version;`

`extern int os_revision;`

The major and minor version of the Operating System currently running. Set by allegro_init(). If Allegro for some reason was not able to retrieve the version of the Operating System, os_version and os_revision will be set to -1. For example: Under Win98 SE (v4.10.2222) os_version will be set to 4 and os_revision to 10.

See also:

See Section 1.7 [os_type], page 3.

See Section 1.9 [os_multitasking], page 4.

1.9 os_multitasking

```
extern int os_multitasking;
```

Set by allegro_init() to either TRUE or FALSE depending on whether your Operating System is multitasking or not.

See also:

See Section 1.7 [os_type], page 3.

See Section 1.8 [os_version], page 3.

1.10 allegro_message

```
void allegro_message(const char *msg, ...);
```

Outputs a message, using a printf() format string. This function must only be used when you aren't in graphics mode, eg. before calling set_gfx_mode(), or after a set_gfx_mode(GFX_TEXT). On platforms that have a text console (DOS and Unix) it will print the string to that console, attempting to work around codepage differences by reducing any accented characters to 7 bit ASCII approximations, and on platforms featuring a windowing system it will bring up a GUI message box.

1.11 set_window_title

```
void set_window_title(const char *name);
```

On platforms that are capable of it, this routine alters the window title for your Allegro program. Note that Allegro cannot set the window title when running in a DOS box under Windows.

See also:

See Section 1.12 [set_window_close_button], page 4.

See Section 1.13 [set_window_close_hook], page 5.

1.12 set_window_close_button

```
int set_window_close_button(int enable);
```

On platforms that are capable of it, this routine disables or enables the window close button for your Allegro program. You can call it before the window is created if you wish. If the close button is successfully disabled, this function returns zero.

On platforms where the close button either does not exist or cannot be disabled, this function returns -1. If this happens, you may wish to use set_window_close_hook() to handle the close event yourself.

When enabling the close button, the function will return the same value for your platform as when disabling. That means it will return non-zero if the button cannot be disabled, even though you are not trying to disable it.

Note that Allegro cannot manipulate the close button of a DOS box in Windows.

See also:

See Section 1.13 [set_window_close_hook], page 5.

See Section 1.11 [set_window_title], page 4.

1.13 set_window_close_hook

```
void set_window_close_hook(void (*proc)());
```

On platforms that have a close button, this routine installs a hook function to handle the close event. In other words, when the user clicks the close button on your program's window, the function you specify here will be called.

This function should not generally attempt to exit the program or save any data itself. The function could be called at any time, and there is usually a risk of conflict with the main thread of the program. Instead, you should set a flag during this function, and test it on a regular basis in the main loop of the program.

Pass NULL to this function to restore the close button's default functionality. On Windows and BeOS, the following message will appear:

Warning: forcing program shutdown may lead to data loss and unexpected results. It is preferable to use the exit command inside the window.

Proceed anyway?

[Yes] [No]

This message will be translated into your selected language if a translation is available in language.dat (see get_config_text()).

If the user clicks [Yes], the program will exit immediately in the same style as Ctrl+Alt+End (see three_finger_flag).

In other operating systems, the program will exit immediately without prompting the user.

Note that Allegro cannot intercept the close button of a DOS box in Windows.

See also:

See Section 1.12 [set_window_close_button], page 4.

See Section 1.11 [set_window_title], page 4.

See Section 3.17 [get_config_text], page 33.

See Section 6.20 [three_finger_flag], page 62.

1.14 desktop_color_depth

```
int desktop_color_depth();
```

On platforms that can run Allegro programs in a window on an existing desktop, returns the currently selected desktop color depth (your program is likely to run faster if it uses this same depth). On platforms where this information is not available or does not apply, returns zero.

See also:

See Section 1.15 [get_desktop_resolution], page 6.

1.15 get_desktop_resolution

```
int get_desktop_resolution(int *width, int *height);
```

On platforms that can run Allegro programs in a window on an existing desktop, this retrieves the current desktop resolution (e.g. you may want to call this function before creating a large window because, with some windowed drivers, a window cannot be created if it is larger than the desktop). Returns zero on success, or a negative number if this information is not available or does not apply, in which case the values stored in width and height are unspecified.

See also:

See Section 1.14 [desktop_color_depth], page 5.

1.16 yield_timeslice

```
void yield_timeslice();
```

On systems that support this, gives up the rest of the current scheduler timeslice. Also known as the "play nice with multitasking" option.

1.17 check_cpu

```
void check_cpu();
```

Detects the CPU type, setting the following global variables. You don't normally need to call this, because allegro_init() will do it for you.

See also:

See Section 1.18 [cpu_vendor], page 6.

See Section 1.19 [cpu_family], page 7.

See Section 1.20 [cpu_model], page 7.

See Section 1.21 [cpu_capabilities], page 7.

1.18 cpu_vendor

```
extern char cpu_vendor[];
```

Contains the CPU vendor name, if known (empty string on non-Intel platforms).

See also:

See Section 1.17 [check_cpu], page 6.

See Section 1.19 [cpu_family], page 7.

See Section 1.20 [cpu_model], page 7.

See Section 1.21 [cpu_capabilities], page 7.

1.19 cpu_family

`extern int cpu_family;`

Contains the Intel CPU type, where applicable: 3=386, 4=486, 5=Pentium, 6=PPro, etc.

See also:
See Section 1.17 [check_cpu], page 6.
See Section 1.18 [cpu_vendor], page 6.
See Section 1.20 [cpu_model], page 7.
See Section 1.21 [cpu_capabilities], page 7.

1.20 cpu_model

`extern int cpu_model;`

Contains the Intel CPU submodel, where applicable. On a 486 (cpu_family=4), zero or one indicates a DX chip, 2 an SX, 3 a 487 (SX) or 486 DX, 4 an SL, 5 an SX2, 7 a DX2 write-back enhanced, 8 a DX4 or DX4 overdrive, 14 a Cyrix, and 15 is unknown. On a Pentium chip (cpu_family=5), 1 indicates a Pentium (510\66, 567\66), 2 is a Pentium P54C, 3 is a Pentium overdrive processor, 5 is a Pentium overdrive for IntelDX4, 14 is a Cyrix, and 15 is unknown.

See also:
See Section 1.17 [check_cpu], page 6.
See Section 1.18 [cpu_vendor], page 6.
See Section 1.19 [cpu_family], page 7.
See Section 1.21 [cpu_capabilities], page 7.

1.21 cpu_capabilities

`extern int cpu_capabilities;`

Contains CPU flags indicating what features are available on the current CPU. The flags can be any combination of these:

```
CPU_ID       - Indicates that the "cpuid" instruction is avail-
able. If this
               is set, then all Allegro CPU variables are 100% reliable,
               otherwise there may be some mistakes.
CPU_FPU      - An x87 FPU is available.
CPU_MMX      - Intel MMX  instruction set is available.
CPU_MMXPLUS  - Intel MMX+ instruction set is available.
```

```
          CPU_SSE      - Intel SSE  instruction set is available.
          CPU_SSE2     - Intel SSE2 instruction set is available.
          CPU_3DNOW    - AMD 3DNow! instruction set is available.
          CPU_ENH3DNOW - AMD Enhanced 3DNow! instruction set is available.
          CPU_CMOV     - Pentium Pro "cmov" instruction is available.
```

You can check for multiple features by OR-ing the flags together. For example, to check if the CPU has an FPU and MMX instructions available, you'd do:

```
          if ((cpu_capabilities & (CPU_FPU | CPU_MMX)) == (CPU_FPU | CPU_MMX))
             printf("CPU has both an FPU and MMX instructions!\n");
```

See also:
See Section 1.17 [check_cpu], page 6.
See Section 1.18 [cpu_vendor], page 6.
See Section 1.19 [cpu_family], page 7.
See Section 1.20 [cpu_model], page 7.
See Section 1.21 [cpu_capabilities], page 7.

2 Unicode routines

Allegro can manipulate and display text using any character values from 0 right up to 2^32-1 (although the current implementation of the grabber can only create fonts using characters up to 2^16-1). You can choose between a number of different text encoding formats, which controls how strings are stored and how Allegro interprets strings that you pass to it. This setting affects all aspects of the system: whenever you see a function that returns a char * type, or that takes a char * as an argument, that text will be in whatever format you have told Allegro to use.

By default, Allegro uses UTF-8 encoded text (U_UTF8). This is a variable-width format, where characters can occupy anywhere from one to six bytes. The nice thing about it is that characters ranging from 0-127 are encoded directly as themselves, so UTF-8 is upwardly compatible with 7 bit ASCII ("Hello, World!" means the same thing regardless of whether you interpret it as ASCII or UTF-8 data). Any character values above 128, such as accented vowels, the UK currency symbol, and Arabic or Chinese characters, will be encoded as a sequence of two or more bytes, each in the range 128-255. This means you will never get what looks like a 7 bit ASCII character as part of the encoding of a different character value, which makes it very easy to manipulate UTF-8 strings.

There are a few editing programs that understand UTF-8 format text files. Alternatively, you can write your strings in plain ASCII or 16 bit Unicode formats, and then use the Allegro textconv program to convert them into UTF-8.

If you prefer to use some other text format, you can set Allegro to work with normal 8 bit ASCII (U_ASCII), or 16 bit Unicode (U_UNICODE) instead, or you can provide some handler functions to make it support whatever other text encoding you like (for example it would be easy to add support for 32 bit UCS-4 characters, or the Chinese GB-code format).

There is some limited support for alternative 8 bit codepages, via the U_ASCII_CP mode. This is very slow, so you shouldn't use it for serious work, but it can be handy as an easy way to convert text between different codepages. By default the U_ASCII_CP mode is set up to reduce text to a clean 7 bit ASCII format, trying to replace any accented vowels with their simpler equivalents (this is used by the allegro_message() function when it needs to print an error report onto a text mode DOS screen). If you want to work with other codepages, you can do this by passing a character mapping table to the set_ucodepage() function.

Note that you can use the Unicode routines before you call install_allegro() or allegro_init(). If you want to work in a text mode other than UTF-8, it is best to set it with set_uformat() just before you call these.

2.1 set_uformat

`void set_uformat(int type);`

Sets the current text encoding format. This will affect all parts of Allegro, wherever you see a function that returns a char *, or takes a char * as a parameter. The type should be one of the values:

```
U_ASCII     - fixed size, 8 bit ASCII characters
U_ASCII_CP  - alternative 8 bit codepage (see set_ucodepage())
U_UNICODE   - fixed size, 16 bit Unicode characters
U_UTF8      - variable size, UTF-8 format Unicode characters
```

Although you can change the text format on the fly, this is not a good idea. Many strings, for example the names of your hardware drivers and any language translations, are loaded when you call allegro_init(), so if you change the encoding format after this, they will be in the wrong format, and things will not work properly. Generally you should only call set_uformat() once, before allegro_init(), and then leave it on the same setting for the duration of your program.

See also:

See Section 2.2 [get_uformat], page 10.

See Section 2.3 [register_uformat], page 10.

See Section 2.4 [set_ucodepage], page 10.

See Section 2.1 [set_uformat], page 9.

See Section 2.8 [uconvert], page 12.

See Section 2.23 [ustrsize], page 17.

See Section 2.12 [ugetc], page 13.

See Section 2.13 [ugetx], page 13.

See Section 2.14 [usetc], page 14.

See Section 2.15 [uwidth], page 14.

See Section 2.16 [ucwidth], page 14.

See Section 2.17 [uisok], page 15.

See Section 2.18 [uoffset], page 15.

See Section 2.19 [ugetat], page 15.

See Section 2.20 [usetat], page 16.
See Section 2.21 [uinsert], page 16.
See Section 2.22 [uremove], page 16.
See Section 1.2 [allegro_init], page 1.

2.2 get_uformat

`int get_uformat(void);`

Returns the currently selected text encoding format.

See also:
See Section 2.1 [set_uformat], page 9.

2.3 register_uformat

```
void register_uformat(int type, int (*u_getc)(const char *s), int
(*u_getx)(char **s), int (*u_setc)(char *s, int c), int (*u_width)(const
char *s), int (*u_cwidth)(int c), int (*u_isok)(int c));
```

Installs a set of custom handler functions for a new text encoding format. The type is the ID code for your new format, which should be a 4-character string as produced by the AL_ID() macro, and which can later be passed to functions like set_uformat() and uconvert(). The function parameters are handlers that implement the character access for your new type: see below for details of these.

See also:
See Section 2.1 [set_uformat], page 9.
See Section 2.8 [uconvert], page 12.
See Section 2.12 [ugetc], page 13.
See Section 2.13 [ugetx], page 13.
See Section 2.14 [usetc], page 14.
See Section 2.15 [uwidth], page 14.
See Section 2.16 [ucwidth], page 14.
See Section 2.17 [uisok], page 15.

2.4 set_ucodepage

```
void set_ucodepage(const unsigned short *table, const unsigned short
*extras);
```

When you select the U_ASCII_CP encoding mode, a set of tables are used to convert between 8 bit characters and their Unicode equivalents. You can use this function to specify a custom set of mapping tables, which allows you to support different 8 bit codepages. The table parameter points to an array of 256 shorts, which contain the Unicode value for each character in your codepage. The extras parameter, if not NULL, points to a list of mapping pairs, which will be used when reducing Unicode data to your codepage. Each pair consists of a Unicode value, followed by the way it should be represented in your codepage. The table is terminated by a zero Unicode value. This allows you to create a many->one mapping, where many different Unicode characters can be represented by a single codepage value (eg. for reducing accented vowels to 7 bit ASCII).

See also:

See Section 2.1 [set_uformat], page 9.

2.5 need_uconvert

`int need_uconvert(const char *s, int type, int newtype);`

Given a pointer to a string, a description of the type of the string, and the type that you would like this string to be converted into, this function tells you whether any conversion is required. No conversion will be needed if type and newtype are the same, or if one type is ASCII, the other is UTF-8, and the string contains only character values less than 128. As a convenience shortcut, you can pass the value U_CURRENT as either of the type parameters, to represent whatever text format is currently selected.

See also:

See Section 2.1 [set_uformat], page 9.

See Section 2.2 [get_uformat], page 10.

See Section 2.7 [do_uconvert], page 11.

See Section 2.8 [uconvert], page 12.

2.6 uconvert_size

`int uconvert_size(const char *s, int type, int newtype);`

Returns the number of bytes that will be required to store the specified string after a conversion from type to newtype, including the zero terminator. The type parameters can use the value U_CURRENT as a shortcut to represent the currently selected encoding format.

See also:

See Section 2.5 [need_uconvert], page 11.

See Section 2.7 [do_uconvert], page 11.

2.7 do_uconvert

`void do_uconvert(const char *s, int type, char *buf, int newtype, int size);`

Converts the specified string from type to newtype, storing at most size bytes into the output buf. The type parameters can use the value U_CURRENT as a shortcut to represent the currently selected encoding format.

See also:

See Section 2.8 [uconvert], page 12.

2.8 uconvert

`char *uconvert(const char *s, int type, char *buf, int newtype, int size);`

Higher level function running on top of do_uconvert(). This function converts the specified string from type to newtype, storing at most size bytes into the output buf, but it checks before doing the conversion, and doesn't bother if the string formats are already the same (either both types are equal, or one is ASCII, the other is UTF-8, and the string contains only 7 bit ASCII characters). If a conversion was performed it returns a pointer to buf, otherwise it returns a copy of s, so you must use the return value rather than assuming that the string will always be moved to buf. As a convenience, if buf is NULL it will convert the string into an internal static buffer. You should be wary of using this feature, though, because that buffer will be overwritten the next time this routine is called, so don't expect the data to persist across any other library calls.

See also:

See Section 2.1 [set_uformat], page 9.

See Section 2.5 [need_uconvert], page 11.

See Section 2.8 [uconvert], page 12.

See Section 2.9 [uconvert_ascii], page 12.

See Section 2.10 [uconvert_toascii], page 12.

2.9 uconvert_ascii

`char *uconvert_ascii(const char *s, char buf[]);`

Helper macro for converting strings from ASCII into the current encoding format. Expands to uconvert(s, U_ASCII, buf, U_CURRENT, sizeof(buf)).

See also:

See Section 2.8 [uconvert], page 12.

2.10 uconvert_toascii

`char *uconvert_toascii(const char *s, char buf[]);`

Helper macro for converting strings from the current encoding format into ASCII. Expands to uconvert(s, U_CURRENT, buf, U_ASCII, sizeof(buf)).

See also:

See Section 2.8 [uconvert], page 12.

2.11 empty_string

`extern char empty_string[];`

You can't just rely on `""` to be a valid empty string in any encoding format. This global buffer contains a number of consecutive zeros, so it will be a valid empty string no matter whether the program is running in ASCII, Unicode, or UTF-8 mode.

2.12 ugetc

`int ugetc(const char *s);`

Low level helper function for reading Unicode text data. Given a pointer to a string in the current encoding format, it returns the next character from the string.

See also:

See Section 2.13 [ugetx], page 13.

See Section 2.14 [usetc], page 14.

See Section 2.15 [uwidth], page 14.

See Section 2.16 [ucwidth], page 14.

See Section 2.17 [uisok], page 15.

2.13 ugetx

`int ugetx(char **s);`

`int ugetxc(const char **s);`

Low level helper function for reading Unicode text data. Given the address of a pointer to a string in the current encoding format, it returns the next character from the string, and advances the pointer to the character after the one just read.

ugetxc is provided for working with pointer-to-pointer-to-const char data.

See also:

See Section 2.12 [ugetc], page 13.

See Section 2.14 [usetc], page 14.

See Section 2.15 [uwidth], page 14.

See Section 2.16 [ucwidth], page 14.

See Section 2.17 [uisok], page 15.

2.14 usetc

`int usetc(char *s, int c);`

Low level helper function for writing Unicode text data. It writes the specified character to the given address in the current encoding format, and returns the number of bytes written.

See also:

See Section 2.12 [ugetc], page 13.

See Section 2.13 [ugetx], page 13.

See Section 2.15 [uwidth], page 14.

See Section 2.16 [ucwidth], page 14.

See Section 2.17 [uisok], page 15.

2.15 uwidth

`int uwidth(const char *s);`

Low level helper function for testing Unicode text data. It returns the number of bytes occupied by the first character of the specified string, in the current encoding format.

See also:

See Section 2.25 [uwidth_max], page 17.

See Section 2.12 [ugetc], page 13.

See Section 2.13 [ugetx], page 13.

See Section 2.14 [usetc], page 14.

See Section 2.16 [ucwidth], page 14.

See Section 2.17 [uisok], page 15.

2.16 ucwidth

`int ucwidth(int c);`

Low level helper function for testing Unicode text data. It returns the number of bytes that would be occupied by the specified character value, when encoded in the current format.

See also:

See Section 2.25 [uwidth_max], page 17.

See Section 2.12 [ugetc], page 13.

See Section 2.13 [ugetx], page 13.

See Section 2.14 [usetc], page 14.

See Section 2.15 [uwidth], page 14.

See Section 2.17 [uisok], page 15.

2.17 uisok

`int uisok(int c);`

Low level helper function for testing Unicode text data. Tests whether the specified value can be correctly encoded in the current format.

See also:

See Section 2.12 [ugetc], page 13.

See Section 2.13 [ugetx], page 13.

See Section 2.14 [usetc], page 14.

See Section 2.15 [uwidth], page 14.

See Section 2.16 [ucwidth], page 14.

2.18 uoffset

`int uoffset(const char *s, int index);`

Returns the offset in bytes from the start of the string to the character at the specified index. If the index is negative, it counts backward from the end of the string, so an index of -1 will return an offset to the last character.

See also:

See Section 2.19 [ugetat], page 15.

See Section 2.20 [usetat], page 16.

See Section 2.21 [uinsert], page 16.

See Section 2.22 [uremove], page 16.

2.19 ugetat

`int ugetat(const char *s, int index);`

Returns the character value at the specified index within the string. A zero index parameter will return the first character of the string. If the index is negative, it counts backward from the end of the string, so an index of -1 will return the last character of the string.

See also:

See Section 2.18 [uoffset], page 15.

See Section 2.20 [usetat], page 16.

See Section 2.21 [uinsert], page 16.

See Section 2.22 [uremove], page 16.

2.20 usetat

`int usetat(char *s, int index, int c);`

Replaces the character at the specified index within the string with value c, handling any adjustments for variable width data (ie. if c encodes to a different width than the previous value at that location). Returns the number of bytes by which the trailing part of the string was moved. If the index is negative, it counts backward from the end of the string.

See also:

See Section 2.18 [uoffset], page 15.

See Section 2.19 [ugetat], page 15.

See Section 2.21 [uinsert], page 16.

See Section 2.22 [uremove], page 16.

2.21 uinsert

`int uinsert(char *s, int index, int c);`

Inserts the character c at the specified index within the string, sliding the rest of the data along to make room. Returns the number of bytes by which the trailing part of the string was moved. If the index is negative, it counts backward from the end of the string.

See also:

See Section 2.18 [uoffset], page 15.

See Section 2.19 [ugetat], page 15.

See Section 2.20 [usetat], page 16.

See Section 2.22 [uremove], page 16.

2.22 uremove

`int uremove(char *s, int index);`

Removes the character at the specified index within the string, sliding the rest of the data back to fill the gap. Returns the number of bytes by which the trailing part of the string was moved. If the index is negative, it counts backward from the end of the string.

See also:

See Section 2.18 [uoffset], page 15.

See Section 2.19 [ugetat], page 15.

See Section 2.20 [usetat], page 16.

See Section 2.21 [uinsert], page 16.

2.23 ustrsize

`int ustrsize(const char *s);`

Returns the size of the specified string in bytes, not including the trailing zero.

See also:

See Section 2.24 [ustrsizez], page 17.

2.24 ustrsizez

`int ustrsizez(const char *s);`

Returns the size of the specified string in bytes, including the trailing zero.

See also:

See Section 2.23 [ustrsize], page 17.

2.25 uwidth_max

`int uwidth_max(int type);`

Low level helper function for working with Unicode text data. Returns the largest number of bytes that one character can occupy in the given encoding format. Pass U_CURRENT to represent the current format.

See also:

See Section 2.15 [uwidth], page 14.

See Section 2.16 [ucwidth], page 14.

2.26 utolower

`int utolower(int c);`

This function returns c, converting it to lower case if it is upper case.

See also:

See Section 2.27 [utoupper], page 18.

See Section 2.12 [ugetc], page 13.

See Section 2.13 [ugetx], page 13.

See Section 2.14 [usetc], page 14.

See Section 2.15 [uwidth], page 14.

See Section 2.16 [ucwidth], page 14.

See Section 2.17 [uisok], page 15.

2.27 utoupper

`int utoupper(int c);`

This function returns c, converting it to upper case if it is lower case.

See also:

See Section 2.26 [utolower], page 17.

See Section 2.12 [ugetc], page 13.

See Section 2.13 [ugetx], page 13.

See Section 2.14 [usetc], page 14.

See Section 2.15 [uwidth], page 14.

See Section 2.16 [ucwidth], page 14.

See Section 2.17 [uisok], page 15.

2.28 uisspace

`int uisspace(int c);`

Returns nonzero if c is whitespace, that is, carriage return, newline, form feed, tab, vertical tab, or space.

See also:

See Section 2.29 [uisdigit], page 18.

See Section 2.12 [ugetc], page 13.

See Section 2.14 [usetc], page 14.

See Section 2.15 [uwidth], page 14.

See Section 2.16 [ucwidth], page 14.

See Section 2.17 [uisok], page 15.

2.29 uisdigit

`int uisdigit(int c);`

Returns nonzero if c is a digit.

See also:

See Section 2.28 [uisspace], page 18.

See Section 2.12 [ugetc], page 13.

See Section 2.14 [usetc], page 14.

See Section 2.15 [uwidth], page 14.

See Section 2.16 [ucwidth], page 14.

See Section 2.17 [uisok], page 15.

2.30 ustrdup

`char *ustrdup(const char *src)`

This functions copies the NULL-terminated string src into a newly allocated area of memory. The memory returned by this call must be freed by the caller. Returns NULL if it cannot allocate space for the duplicated string.

See also:

See Section 2.31 [_ustrdup], page 19.

See Section 2.8 [uconvert], page 12.

See Section 2.23 [ustrsize], page 17.

See Section 2.24 [ustrsizez], page 17.

2.31 _ustrdup

`char *_ustrdup(const char *src, void* (*malloc_func) (size_t))`

Does the same as ustrdup(), but allows the user to specify his own memory allocater function.

See also:

See Section 2.30 [ustrdup], page 19.

See Section 2.8 [uconvert], page 12.

See Section 2.23 [ustrsize], page 17.

See Section 2.24 [ustrsizez], page 17.

2.32 ustrcpy

`char *ustrcpy(char *dest, const char *src);`

This function copies src (including the terminating NULL character) into dest. The return value is the value of dest.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.33 [ustrzcpy], page 19.

See Section 2.38 [ustrncpy], page 21.

2.33 ustrzcpy

```
char *ustrzcpy(char *dest, int size, const char *src);
```

This function copies src (including the terminating NULL character) into dest, whose length in bytes is specified by size and which is guaranteed to be NULL-terminated. The return value is the value of dest.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.32 [ustrcpy], page 19.

See Section 2.39 [ustrzncpy], page 21.

2.34 ustrcat

```
char *ustrcat(char *dest, const char *src);
```

This function concatenates src to the end of dest. The return value is the value of dest.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.35 [ustrzcat], page 20.

See Section 2.40 [ustrncat], page 22.

2.35 ustrzcat

```
char *ustrzcat(char *dest, int size, const char *src);
```

This function concatenates src to the end of dest, whose length in bytes is specified by size and which is guaranteed to be NULL-terminated. The return value is the value of dest.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.34 [ustrcat], page 20.

See Section 2.41 [ustrzncat], page 22.

2.36 ustrlen

```
int ustrlen(const char *s);
```

This function returns the number of characters in s. Note that this doesn't have to equal the string's size in bytes.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.23 [ustrsize], page 17.

See Section 2.24 [ustrsizez], page 17.

2.37 ustrcmp

`int ustrcmp(const char *s1, const char *s2);`

This function compares s1 and s2. Returns zero if the strings are equal, a positive number if s1 comes after s2 in the ASCII collating sequence, else a negative number.

See also:
See Section 2.8 [uconvert], page 12.
See Section 2.23 [ustrsize], page 17.
See Section 2.24 [ustrsizez], page 17.
See Section 2.42 [ustrncmp], page 22.
See Section 2.43 [ustricmp], page 23.

2.38 ustrncpy

`char *ustrncpy(char *dest, const char *src, int n);`

This function is like ustrcpy() except that no more than n characters from src are copied into dest. If src is shorter than n characters, NULL characters are appended to dest as padding until n characters have been written. Note that if src is longer than n characters, dest will not be NULL-terminated. The return value is the value of dest.

See also:
See Section 2.8 [uconvert], page 12.
See Section 2.32 [ustrcpy], page 19.
See Section 2.39 [ustrzncpy], page 21.

2.39 ustrzncpy

`char *ustrzncpy(char *dest, int size, const char *src, int n);`

This function is like ustrzcpy() except that no more than n characters from src are copied into dest. If src is shorter than n characters, NULL characters are appended to dest as padding until n characters have been written. Note that dest is guaranteed to be NULL-terminated. The return value is the value of dest.

See also:
See Section 2.8 [uconvert], page 12.
See Section 2.33 [ustrzcpy], page 19.

See Section 2.38 [ustrncpy], page 21.

2.40 ustrncat

```
char *ustrncat(char *dest, const char *src, int n);
```

This function is like ustrcat() except that no more than n characters from src are appended to the end of dest. If the terminating NULL character in src is reached before n characters have been written, the NULL character is copied, but no other characters are written. If n characters are written before a terminating NULL is encountered, the function appends its own NULL character to dest, so that n+1 characters are written. The return value is the value of dest.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.34 [ustrcat], page 20.

See Section 2.41 [ustrzncat], page 22.

2.41 ustrzncat

```
char *ustrzncat(char *dest, int size, const char *src, int n);
```

This function is like ustrzcat() except that no more than n characters from src are appended to the end of dest. If the terminating NULL character in src is reached before n characters have been written, the NULL character is copied, but no other characters are written. Note that dest is guaranteed to be NULL-terminated. The return value is the value of dest.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.35 [ustrzcat], page 20.

See Section 2.40 [ustrncat], page 22.

2.42 ustrncmp

```
int ustrncmp(const char *s1, const char *s2, int n);
```

This function compares up to n characters of s1 and s2. Returns zero if the sub-strings are equal, a positive number if s1 comes after s2 in the ASCII collating sequence, else a negative number.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.23 [ustrsize], page 17.

See Section 2.24 [ustrsizez], page 17.

See Section 2.37 [ustrcmp], page 21.

See Section 2.43 [ustricmp], page 23.

2.43 ustricmp

`int ustricmp(const char *s1, const char *s2);`

This function compares s1 and s2, ignoring case.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.23 [ustrsize], page 17.

See Section 2.24 [ustrsizez], page 17.

See Section 2.37 [ustrcmp], page 21.

See Section 2.42 [ustrncmp], page 22.

2.44 ustrlwr

`char *ustrlwr(char *s);`

This function replaces all upper case letters in s with lower case letters.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.26 [utolower], page 17.

See Section 2.45 [ustrupr], page 23.

2.45 ustrupr

`char *ustrupr(char *s);`

This function replaces all lower case letters in s with upper case letters.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.26 [utolower], page 17.

See Section 2.44 [ustrlwr], page 23.

2.46 ustrchr

`char *ustrchr(const char *s, int c);`

This function returns a pointer to the first occurrence of c in s, or NULL if no match was found. Note that if c is NULL, this will return a pointer to the end of the string.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.47 [ustrrchr], page 24.

See Section 2.48 [ustrstr], page 24.

See Section 2.49 [ustrpbrk], page 24.

See Section 2.50 [ustrtok], page 25.

2.47 ustrrchr

`char *ustrrchr(const char *s, int c);`

This function returns a pointer to the last occurrence of c in s, or NULL if no match was found.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.46 [ustrchr], page 23.

See Section 2.48 [ustrstr], page 24.

See Section 2.49 [ustrpbrk], page 24.

See Section 2.50 [ustrtok], page 25.

2.48 ustrstr

`char *ustrstr(const char *s1, const char *s2);`

This function finds the first occurence of s2 in s1. Returns a pointer within s1, or NULL if s2 wasn't found.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.46 [ustrchr], page 23.

See Section 2.47 [ustrrchr], page 24.

See Section 2.49 [ustrpbrk], page 24.

See Section 2.50 [ustrtok], page 25.

2.49 ustrpbrk

`char *ustrpbrk(const char *s, const char *set);`

This function finds the first character in s that matches any character in set. Returns a pointer to the first match, or NULL if none are found.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.46 [ustrchr], page 23.

See Section 2.47 [ustrrchr], page 24.

See Section 2.48 [ustrstr], page 24.

See Section 2.50 [ustrtok], page 25.

2.50 ustrtok

`char *ustrtok(char *s, const char *set);`

This function retrieves tokens from s which are delimited by characters from set. To initiate the search, pass the string to be searched as s. For the remaining tokens, pass NULL instead. Returns a pointer to the token, or NULL if no more are found. Warning: Since ustrtok alters the string it is parsing, you should always copy the string to a temporary buffer before parsing it. Also, this function is not reentrant (ie. you cannot parse two strings at the same time).

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.46 [ustrchr], page 23.

See Section 2.47 [ustrrchr], page 24.

See Section 2.48 [ustrstr], page 24.

See Section 2.49 [ustrpbrk], page 24.

See Section 2.51 [ustrtok_r], page 25.

2.51 ustrtok_r

`char *ustrtok_r(char *s, const char *set, char **last);`

Reentrant version of ustrtok. The last parameter is used to keep track of where the parsing is up to and must be a pointer to a char * variable allocated by the user that remains the same while parsing the same string.

See also:

See Section 2.50 [ustrtok], page 25.

2.52 uatof

`double uatof(const char *s);`

Convert as much of the string as possible to an equivalent double precision real number. This function is almost like 'ustrtod(s, NULL)'. Returns the equivalent value, or zero if the string does not represent a number.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.53 [ustrtol], page 26.

See Section 2.54 [ustrtod], page 26.

2.53 ustrtol

`long ustrtol(const char *s, char **endp, int base);`

This function converts the initial part of s to a signed integer, which is returned as a value of type 'long int', setting *endp to point to the first unused character, if endp is not a NULL pointer. The base argument indicates what base the digits (or letters) should be treated as. If base is zero, the base is determined by looking for '0x', '0X', or '0' as the first part of the string, and sets the base used to 16, 16, or 8 if it finds one. The default base is 10 if none of those prefixes are found.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.54 [ustrtod], page 26.

See Section 2.52 [uatof], page 25.

2.54 ustrtod

`double ustrtod(const char *s, char **endp);`

This function converts as many characters of s that look like a floating point number into one, and sets *endp to point to the first unused character, if endp is not a NULL pointer.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.53 [ustrtol], page 26.

See Section 2.52 [uatof], page 25.

2.55 ustrerror

`const char *ustrerror(int err);`

This function returns a string that describes the error code 'err', which normally comes from the variable 'errno'. Returns a pointer to a static string that should not be modified or free'd. If you make subsequent calls to ustrerror, the string might be overwritten.

See also:

See Section 2.8 [uconvert], page 12.

See Section 1.6 [allegro_error], page 2.

2.56 usprintf

`int usprintf(char *buf, const char *format, ...);`

This function writes formatted data into the output buffer. A NULL character is written to mark the end of the string. Returns the number of characters written, not including the terminating NULL character.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.57 [uszprintf], page 27.

See Section 2.58 [uvsprintf], page 27.

2.57 uszprintf

`int uszprintf(char *buf, int size, const char *format, ...);`

This function writes formatted data into the output buffer, whose length in bytes is specified by size and which is guaranteed to be NULL terminated. Returns the number of characters that would have been written without eventual truncation (like with usprintf), not including the terminating NULL character.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.56 [usprintf], page 26.

See Section 2.59 [uvszprintf], page 27.

2.58 uvsprintf

`int uvsprintf(char *buf, const char *format, va_list args);`

This is like usprintf(), but you pass the variable argument list directly, instead of the arguments themselves.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.56 [usprintf], page 26.

See Section 2.59 [uvszprintf], page 27.

2.59 uvszprintf

`int uvszprintf(char *buf, int size, const char *format, va_list args);`

This is like uszprintf(), but you pass the variable argument list directly, instead of the arguments themselves.

See also:

See Section 2.8 [uconvert], page 12.

See Section 2.57 [uszprintf], page 27.

See Section 2.58 [uvsprintf], page 27.

3 Configuration routines

Various parts of Allegro, such as the sound routines and the load_joystick_data() function, require some configuration information. This data is stored in text files as a collection of "variable=value" lines, along with comments that begin with a '#' character and continue to the end of the line. The configuration file may optionally be divided into sections, which begin with a "[sectionname]" line. Each section has a unique namespace, to prevent variable name conflicts, but any variables that aren't in a section are considered to belong to all the sections simultaneously.

By default the configuration data is read from a file called allegro.cfg, which can be located either in the same directory as the program executable, or the directory pointed to by the ALLEGRO environment variable. Under Unix, it also checks for ~/allegro.cfg, ~/.allegrorc, /etc/allegro.cfg, and /etc/allegrorc, in that order; under BeOS only the last two are also checked. If you don't like this approach, you can specify any filename you like, or use a block of binary configuration data provided by your program (which could for example be loaded from a datafile).

You can store whatever custom information you like in the config file, along with the standard variables that are used by Allegro (see below).

3.1 set_config_file

```
void set_config_file(const char *filename);
```

Sets the configuration file to be used by all subsequent config functions. If you don't call this function, Allegro will use the default allegro.cfg file, looking first in the same directory as your program and then in the directory pointed to by the ALLEGRO environment variable.

All pointers returned by previous calls to get_config_string() and other related functions are invalidated when you call this function!

See also:

See Section 3.2 [set_config_data], page 28.

See Section 3.3 [override_config_file], page 29.

See Section 3.5 [push_config_state], page 29.

See Section 3.23 [standard config variables], page 36.

See Section 3.18 [set_config_string], page 34.

See Section 3.11 [get_config_string], page 31.

3.2 set_config_data

```
void set_config_data(const char *data, int length);
```

Specifies a block of data to be used by all subsequent config functions, which you have already loaded from disk (eg. as part of some more complicated format of your own, or in a grabber datafile). This routine makes a copy of the information, so you can safely free the data after calling it.

See also:

See Section 3.1 [set_config_file], page 28.

See Section 3.4 [override_config_data], page 29.

See Section 3.5 [push_config_state], page 29.

See Section 3.23 [standard config variables], page 36.

See Section 3.18 [set_config_string], page 34.

See Section 3.11 [get_config_string], page 31.

3.3 override_config_file

`void override_config_file(const char *filename);`

Specifies a file containing config overrides. These settings will be used in addition to the parameters in the main config file, and where a variable is present in both files this version will take priority. This can be used by application programmers to override some of the config settings from their code, while still leaving the main config file free for the end user to customise. For example, you could specify a particular sample frequency and IBK instrument file, but the user could still use an allegro.cfg file to specify the port settings and irq numbers.

See also:

See Section 3.4 [override_config_data], page 29.

See Section 3.1 [set_config_file], page 28.

3.4 override_config_data

`void override_config_data(const char *data, int length);`

Version of override_config_file() which uses a block of data that has already been read into memory.

See also:

See Section 3.3 [override_config_file], page 29.

See Section 3.2 [set_config_data], page 28.

3.5 push_config_state

`void push_config_state();`

Pushes the current configuration state (filename, variable values, etc). onto an internal stack, allowing you to select some other config source and later restore the current settings by calling pop_config_state(). This function is mostly intended for internal use by other library functions, for example when you specify a config filename to the save_joystick_data() function, it pushes the config state before switching to the file you specified.

See also:

See Section 3.6 [pop_config_state], page 30.

See Section 3.1 [set_config_file], page 28.

3.6 pop_config_state

```
void pop_config_state();
```

Pops a configuration state previously stored by push_config_state(), replacing the current config source with it.

See also:

See Section 3.5 [push_config_state], page 29.

3.7 flush_config_file

```
void flush_config_file();
```

Writes the current config file to disk if the contents have changed since it was loaded or since the latest call to the function.

See also:

See Section 3.1 [set_config_file], page 28.

See Section 3.3 [override_config_file], page 29.

See Section 3.5 [push_config_state], page 29.

3.8 reload_config_texts

```
void reload_config_texts(const char *new_language);
```

Reloads the translated strings returned by get_config_text. This is useful to switch to another language in your program at runtime. If you want to modify the [system] language configuration variable yourself, or you have switched configuration files, you will want to pass NULL to just reload whatever language is currently selected. Or you can pass a string containing the two letter code of the language you desire to switch to, and the function will modify the language variable. After you call this function, the previously returned pointers of get_config_text will be invalid.

See also:

See Section 3.17 [get_config_text], page 33.

See Section 3.11 [get_config_string], page 31.

See Section 3.23 [standard config variables], page 36.

3.9 hook_config_section

`void hook_config_section(const char *section, int (*intgetter)(const char *name, int def), const char *(*stringgetter)(const char *name, const char *def), void (*stringsetter)(const char *name, const char *value));`

Takes control of the specified config file section, so that your hook functions will be used to manipulate it instead of the normal disk file access. If both the getter and setter functions are NULL, a currently present hook will be unhooked. Hooked functions have the highest priority. If a section is hooked, the hook will always be called, so you can also hook a '#' section: even override_config_file() cannot override a hooked section.

See also:

See Section 3.10 [config_is_hooked], page 31.

3.10 config_is_hooked

`int config_is_hooked(const char *section);`

Returns TRUE if the specified config section has been hooked.

See also:

See Section 3.9 [hook_config_section], page 31.

3.11 get_config_string

`const char *get_config_string(const char *section, const char *name, const char *def);`

Retrieves a string variable from the current config file. If the named variable cannot be found, or its entry in the config file is empty, the value of def is returned. The section name may be set to NULL to read variables from the root of the file, or used to control which set of parameters (eg. sound or joystick) you are interested in reading.

See also:

See Section 3.1 [set_config_file], page 28.

See Section 3.18 [set_config_string], page 34.

See Section 3.16 [get_config_argv], page 33.

See Section 3.14 [get_config_float], page 32.

See Section 3.13 [get_config_hex], page 32.

See Section 3.12 [get_config_int], page 31.

See Section 3.15 [get_config_id], page 33.

3.12 get_config_int

`int get_config_int(const char *section, const char *name, int def);`

Reads an integer variable from the current config file. See the comments about get_config_string().

See also:

See Section 3.1 [set_config_file], page 28.

See Section 3.19 [set_config_int], page 34.

See Section 3.11 [get_config_string], page 31.

See Section 3.16 [get_config_argv], page 33.

See Section 3.14 [get_config_float], page 32.

See Section 3.13 [get_config_hex], page 32.

See Section 3.15 [get_config_id], page 33.

3.13 get_config_hex

`int get_config_hex(const char *section, const char *name, int def);`

Reads an integer variable from the current config file, in hexadecimal format. See the comments about get_config_string().

See also:

See Section 3.1 [set_config_file], page 28.

See Section 3.20 [set_config_hex], page 35.

See Section 3.11 [get_config_string], page 31.

See Section 3.16 [get_config_argv], page 33.

See Section 3.14 [get_config_float], page 32.

See Section 3.12 [get_config_int], page 31.

See Section 3.15 [get_config_id], page 33.

3.14 get_config_float

`float get_config_float(const char *section, const char *name, float def);`

Reads a floating point variable from the current config file. See the comments about get_config_string().

See also:

See Section 3.1 [set_config_file], page 28.

See Section 3.21 [set_config_float], page 35.

See Section 3.11 [get_config_string], page 31.

See Section 3.16 [get_config_argv], page 33.

See Section 3.13 [get_config_hex], page 32.

See Section 3.12 [get_config_int], page 31.

See Section 3.15 [get_config_id], page 33.

3.15 get_config_id

`int get_config_id(const char *section, const char *name, int def);`

Reads a 4-letter driver ID variable from the current config file. See the comments about get_config_string().

See also:

See Section 3.1 [set_config_file], page 28.

See Section 3.22 [set_config_id], page 35.

See Section 3.11 [get_config_string], page 31.

See Section 3.16 [get_config_argv], page 33.

See Section 3.14 [get_config_float], page 32.

See Section 3.13 [get_config_hex], page 32.

See Section 3.12 [get_config_int], page 31.

3.16 get_config_argv

`char **get_config_argv(const char *section, const char *name, int *argc);`

Reads a token list (words separated by spaces) from the current config file, returning a an argv style argument list, and setting argc to the number of tokens (unlike argc/argv, this list is zero based). Returns NULL and sets argc to zero if the variable is not present. The token list is stored in a temporary buffer that will be clobbered by the next call to get_config_argv(), so the data should not be expected to persist.

See also:

See Section 3.1 [set_config_file], page 28.

See Section 3.11 [get_config_string], page 31.

See Section 3.14 [get_config_float], page 32.

See Section 3.13 [get_config_hex], page 32.

See Section 3.12 [get_config_int], page 31.

See Section 3.15 [get_config_id], page 33.

3.17 get_config_text

`const char *get_config_text(const char *msg);`

This function is primarily intended for use by internal library code, but it may perhaps be helpful to application programmers as well. It uses the language.dat or XXtext.cfg files (where XX is a language code) to look up a translated version of the parameter in the currently selected language, returning a suitable translation if one can be found or a copy of the parameter if nothing else is available. This is basically the same thing as calling get_config_string() with [language] as the section, msg as the variable name, and msg as the default value, but it contains some special code to handle Unicode format conversions. The msg parameter is always given in ASCII format, but the returned string will be converted into the current text encoding, with memory being allocated as required, so you can assume that this pointer will persist without having to manually allocate storage space for each string.

See also:
See Section 3.11 [get_config_string], page 31.
See Section 3.8 [reload_config_texts], page 30.
See Section 3.23 [standard config variables], page 36.

3.18 set_config_string

```
void set_config_string(const char *section, const char *name, const char
*val);
```

Writes a string variable to the current config file, replacing any existing value it may have, or removes the variable if val is NULL. The section name may be set to NULL to write the variable to the root of the file, or used to control which section the variable is inserted into. The altered file will be cached in memory, and not actually written to disk until you call allegro_exit(). Note that you can only write to files in this way, so the function will have no effect if the current config source was specified with set_config_data() rather than set_config_file().

As a special case, variable or section names that begin with a '#' character are treated specially and will not be read from or written to the disk. Addon packages can use this to store version info or other status information into the config module, from where it can be read with the get_config_string() function.

See also:
See Section 3.1 [set_config_file], page 28.
See Section 3.11 [get_config_string], page 31.
See Section 3.21 [set_config_float], page 35.
See Section 3.20 [set_config_hex], page 35.
See Section 3.19 [set_config_int], page 34.
See Section 3.22 [set_config_id], page 35.

3.19 set_config_int

```
void set_config_int(const char *section, const char *name, int val);
```

Writes an integer variable to the current config file. See the comments about set_config_string().

See also:
See Section 3.1 [set_config_file], page 28.
See Section 3.12 [get_config_int], page 31.
See Section 3.18 [set_config_string], page 34.
See Section 3.21 [set_config_float], page 35.
See Section 3.20 [set_config_hex], page 35.
See Section 3.22 [set_config_id], page 35.

3.20 set_config_hex

```
void set_config_hex(const char *section, const char *name, int val);
```

Writes an integer variable to the current config file, in hexadecimal format. See the comments about set_config_string().

See also:
See Section 3.1 [set_config_file], page 28.
See Section 3.13 [get_config_hex], page 32.
See Section 3.18 [set_config_string], page 34.
See Section 3.21 [set_config_float], page 35.
See Section 3.19 [set_config_int], page 34.
See Section 3.22 [set_config_id], page 35.

3.21 set_config_float

```
void set_config_float(const char *section, const char *name, float val);
```

Writes a floating point variable to the current config file. See the comments about set_config_string().

See also:
See Section 3.1 [set_config_file], page 28.
See Section 3.14 [get_config_float], page 32.
See Section 3.18 [set_config_string], page 34.
See Section 3.20 [set_config_hex], page 35.
See Section 3.19 [set_config_int], page 34.
See Section 3.22 [set_config_id], page 35.

3.22 set_config_id

```
void set_config_id(const char *section, const char *name, int val);
```

Writes a 4-letter driver ID variable to the current config file. See the comments about set_config_string().

See also:

See Section 3.1 [set_config_file], page 28.

See Section 3.15 [get_config_id], page 33.

See Section 3.18 [set_config_string], page 34.

See Section 3.21 [set_config_float], page 35.

See Section 3.20 [set_config_hex], page 35.

See Section 3.19 [set_config_int], page 34.

3.23 standard config variables

Allegro uses these standard variables from the configuration file:

- [system]
  Section containing general purpose variables:
  - system = x
    Specifies which system driver to use. This is currently only useful on Linux, for choosing between the XWindows ("XWIN") or console ("LNXC") modes.
  - keyboard = x
    Specifies which keyboard layout to use. The parameter is the name of a keyboard mapping file produced by the keyconf utility, and can either be a fully qualified file path or a basename like "us" or "uk". If the latter, Allegro will look first for a separate config file with that name (eg. "uk.cfg") and then for an object with that name in the keyboard.dat file (eg. "UK_CFG"). The config file or keyboard.dat file can be stored in the same directory as the program, or in the location pointed to by the ALLEGRO environment variable. Look in the keyboard.dat file to see what mappings are currently available.
  - language = x
    Specifies which language file to use for error messages and other bits of system text. The parameter is the name of a translation file, and can either be a fully qualified file path or a basename like "en" or "sp". If the latter, Allegro will look first for a separate config file with a name in the form "entext.cfg", and then for an object with that name in the language.dat file (eg. "ENTEXT_CFG"). The config file or language.dat file can be stored in the same directory as the program, or in the location pointed to by the ALLEGRO environment variable. Look in the language.dat file to see which mappings are currently available.
  - menu_opening_delay = x
    Sets how long the menus take to auto-open. The time is given in milliseconds (default is 300). Specifying -1 will disable the auto-opening feature.
  - dga_mouse = x
    X only: disable to work around a bug in some X servers' DGA modes, concerning the mouse. Default is on; enable the workaround by setting the variable to "0".
  - dga_centre = x
    X only: instructs the DGA driver to centre the Allegro screen if the actual screen resolution is higher than Allegro's. Default is on; disable this feature by setting the variable to "0".

  - dga_clear = x
    X only: instructs the DGA driver to clear visible video memory on startup. Default is on; disable this feature by setting the variable to "0".
- [graphics]
  Section containing graphics configuration information, using the variables:
  - gfx_card = x
    Specifies which graphics driver to use when the program requests GFX_AUTODETECT. Multiple possible drivers can be suggested with extra lines in the form "gfx_card1 = x", "gfx_card2 = x", etc, or you can specify different drivers for each mode and color depth with variables in the form "gfx_card_24bpp = x", "gfx_card_640x480x16 = x", etc.
  - gfx_cardw = x
    Specifies which graphics driver to use when the program requests GFX_AUTODETECT_WINDOWED. This variable functions exactly like gfx_card in all other respects. If it is not set, Allegro will look for the gfx_card variable.
  - vbeaf_driver = x
    DOS and Linux only: specifies where to look for the VBE/AF driver (vbeaf.drv). If this variable is not set, Allegro will look in the same directory as the program, and then fall back on the standard locations (c:\ for DOS, /usr/local/lib, /usr/lib, /lib, and / for Linux, or the directory specified with the VBEAF_PATH environment variable).
  - framebuffer = x
    Linux only: specifies what device file to use for the fbcon driver. If this variable is not set, Allegro checks the FRAMEBUFFER environment variable, and then defaults to /dev/fb0.
  - force_centering = x
    Unix/X11 only: specifies whether to force window centering in fullscreen mode when the XWFS driver is used (yes or no). Enabling this setting may cause some artifacts to appear on KDE desktops.
  - disable_direct_updating = x
    Windows only: specifies whether to disable direct updating when the GFX_DIRECTX_WIN driver is used in color conversion mode (yes or no). Direct updating can cause artifacts to be left on the desktop when the window is moved or minimized; disabling it results in a significant performance loss.
- [mouse]
  Section containing mouse configuration information, using the variables:
  - mouse = x
    Mouse driver type. Available DOS drivers are:

    ```
         MICK - mickey mode driver (normally the best)
         I33  - int 0x33 callback driver
         POLL - timer polling (for use under NT)
    ```

    Linux console mouse drivers are:

```
MS   - Microsoft serial mouse
IMS  - Microsoft serial mouse with Intellimouse extension
LPS2 - PS2 mouse
LIPS - PS2 mouse with Intellimouse extension
GPMD - GPM repeater data (Mouse Systems protocol)
```

  - num_buttons = x
    Sets the number of mouse buttons viewed by Allegro. You don't normally need to set this variable because Allegro will autodetect it. You can only use it to restrict the set of actual mouse buttons.
  - emulate_three = x
    Sets whether to emulate a third mouse button by detecting chords of the left and right buttons (yes or no). Defaults to yes if you have a two button mouse, no otherwise.
  - mouse_device = x
    Linux only: specifies the name of the mouse device file (eg. /dev/mouse).
  - mouse_accel_factor = x
    Windows only: specifies the mouse acceleration factor. Defaults to 1. Set it to 0 in order to disable mouse acceleration. 2 accelerates twice as much as 1.
- [sound]
  Section containing sound configuration information, using the variables:
  - digi_card = x
    Sets the driver to use for playing digital samples.
  - midi_card = x
    Sets the driver to use for MIDI music.
  - digi_input_card = x
    Sets the driver to use for digital sample input.
  - midi_input_card = x
    Sets the driver to use for MIDI data input.
  - digi_voices = x
    Specifies the minimum number of voices to reserve for use by the digital sound driver. How many are possible depends on the driver.
  - midi_voices = x
    Specifies the minimum number of voices to reserve for use by the MIDI sound driver. How many are possible depends on the driver.
  - digi_volume = x
    Sets the volume for digital sample playback, from 0 to 255.
  - midi_volume = x
    Sets the volume for midi music playback, from 0 to 255.
  - quality = x
    Controls the sound quality vs. performance tradeoff for the sample mixing code. This can be set to any of the values:

```
0 - fast mixing of 8 bit data into 16 bit buffers
```

```
1 - true 16 bit mixing (requires a 16 bit stereo soundcard)
2 - interpolated 16 bit mixing
```

- flip_pan = x
  Toggling this between 0 and 1 reverses the left/right panning of samples, which might be needed because some SB cards (including mine :-) get the stereo image the wrong way round.
- sound_freq = x
  DOS, Unix and BeOS: sets the sample frequency. With the SB driver, possible rates are 11906 (any), 16129 (any), 22727 (SB 2.0 and above), and 45454 (only on SB 2.0 or SB16, not the stereo SB Pro driver). On the ESS Audiodrive, possible rates are 11363, 17046, 22729, or 44194. On the Ensoniq Soundscape, possible rates are 11025, 16000, 22050, or 48000. On the Windows Sound System, possible rates are 11025, 22050, 44100, or 48000. Don't worry if you set some other number by mistake: Allegro will automatically round it to the closest supported frequency.
- sound_bits = x
  Unix and BeOS: sets the preferred number of bits (8 or 16).
- sound_stereo = x
  Unix and BeOS: selects mono or stereo output (0 or 1).
- sound_port = x
  DOS only: sets the soundcard port address (this is usually 220).
- sound_dma = x
  DOS only: sets the soundcard DMA channel (this is usually 1).
- sound_irq = x
  DOS only: sets the soundcard IRQ number (this is usually 7).
- fm_port = x
  DOS only: sets the port address of the OPL synth (this is usually 388).
- mpu_port = x
  DOS only: sets the port address of the MPU-401 MIDI interface (this is usually 330).
- mpu_irq = x
  DOS only: sets the IRQ for the MPU-401 (this is usually the same as sound_irq).
- ibk_file = x
  DOS only: specifies the name of a .IBK file which will be used to replace the standard Adlib patch set.
- ibk_drum_file = x
  DOS only: specifies the name of a .IBK file which will be used to replace the standard set of Adlib percussion patches.
- oss_driver = x
  Unix only: sets the OSS device driver name. Usually /dev/dsp or /dev/audio, but could be a particular device (e.g. /dev/dsp2).
- oss_numfrags = x
  oss_fragsize = x
  Unix only: sets number of OSS driver fragments (buffers) and size of each buffer in samples. Buffers are filled with data in the interrupts where interval between

subsequent interrupts is not less than 10 ms. If hardware can play all information from buffers faster than 10 ms, then there will be clicks, when hardware have played all data and library has not prepared new data yet. On the other hand, if it takes too long for device driver to play data from all buffers, then there will be delays between action which triggers sound and sound itself.

- oss_midi_driver = x
  Unix only: sets the OSS MIDI device name. Usually /dev/sequencer.
- oss_mixer_driver = x
  Unix only: sets the OSS mixer device name. Usually /dev/mixer.
- esd_server = x
  Unix only: where to find the ESD (Enlightened Sound Daemon) server.
- alsa_card = x
  alsa_pcmdevice = x
  Unix only: paramaters for the ALSA sound driver system.
- alsa_numfrags = x
  Unix only: number of ALSA driver fragments (buffers).
- alsa_fragsize = x
  Unix only: size of each ALSA fragment, in samples.
- alsa_rawmidi_card = x
  Unix only: ALSA card to use for midi output.
- alsa_rawmidi_device = x
  Unix only: ALSA rawmidi device to use for output.
- be_midi_quality = x
  BeOS only: system MIDI synthesizer instruments quality. 0 uses low quality 8-bit 11 kHz samples, 1 uses 16-bit 22 kHz samples.
- be_midi_freq = x
  BeOS only: MIDI sample mixing frequency in Hz. Can be 11025, 22050 or 44100.
- be_midi_interpolation = x
  BeOS only: specifies the MIDI samples interpolation method. 0 doesn't interpolate, it's fast but has the worst quality; 1 does a fast interpolation with better performances, but it's a bit slower than the previous method; 2 does a linear interpolation between samples, it is the slowest method but gives the best performances.
- be_midi_reverb = x
  BeOS only: reverberation intensity, from 0 to 5. 0 disables it, 5 is the strongest one.
- patches = x
  Specifies where to find the sample set for the DIGMID driver. This can either be a Gravis style directory containing a collection of .pat files and a default.cfg index, or an Allegro datafile produced by the pat2dat utility. If this variable is not set, Allegro will look either for a default.cfg or patches.dat file in the same directory as the program, the directory pointed to by the ALLEGRO environment variable, and the standard GUS directory pointed to by the ULTRASND environment variable.

- [midimap]
  If you are using the SB MIDI output or MPU-401 drivers with an external synthesiser

that is not General MIDI compatible, you can use the midimap section of the config file to specify a patch mapping table for converting GM patch numbers into whatever bank and program change messages will select the appropriate sound on your synth. This is a real piece of self-indulgence. I have a Yamaha TG500, which has some great sounds but no GM patch set, and I just had to make it work somehow...

This section consists of a set of lines in the form:

- p<n> = bank0 bank1 prog pitch
  With this statement, n is the GM program change number (1-128), bank0 and bank1 are the two bank change messages to send to your synth (on controllers #0 and #32), prog is the program change message to send to your synth, and pitch is the number of semitones to shift everything that is played with that sound. Setting the bank change numbers to -1 will prevent them from being sent.

  For example, the line:

  p36 = 0 34 9 12

  specifies that whenever GM program 36 (which happens to be a fretless bass) is selected, Allegro should send a bank change message #0 with a parameter of 0, a bank change message #32 with a parameter of 34, a program change with a parameter of 9, and then should shift everything up by an octave.

- [joystick]
  Section containing joystick configuration information, using the variables:

  - joytype = x
    Specifies which joystick driver to use when the program requests JOY_TYPE_AUTODETECT.
  - joystick_device = x
    BeOS only: specifies the name of the joystick device to be used. First device found is used by default.
  - throttle_axis = x
    Linux only: sets which axis number the throttle is located at. This variable will be used for every detected joystick. If you want to specify the axis number for each joystick individually, use variables of the form throttle_axis_n, where n is the joystick number.

See also:
See Section 3.1 [set_config_file], page 28.
See Section 3.18 [set_config_string], page 34.
See Section 3.11 [get_config_string], page 31.

4 Mouse routines

4.1 install_mouse

```
int install_mouse();
```

Installs the Allegro mouse handler. You must do this before using any other mouse functions. Returns -1 on failure, otherwise the number of buttons on the mouse.

See also:

See Section 4.2 [remove_mouse], page 42.

See Section 4.3 [poll_mouse], page 42.

See Section 4.5 [mouse_x], page 43.

See Section 4.7 [show_mouse], page 44.

See Section 4.18 [get_mouse_mickeys], page 47.

See Section 4.12 [position_mouse], page 45.

See Section 4.14 [set_mouse_range], page 46.

See Section 4.15 [set_mouse_speed], page 46.

See Section 3.23 [standard config variables], page 36.

4.2 remove_mouse

```
void remove_mouse();
```

Removes the mouse handler. You don't normally need to bother calling this, because allegro_exit() will do it for you.

See also:

See Section 4.1 [install_mouse], page 41.

See Section 1.3 [allegro_exit], page 1.

4.3 poll_mouse

```
int poll_mouse();
```

Wherever possible, Allegro will read the mouse input asynchronously (ie. from inside an interrupt handler), but on some platforms that may not be possible, in which case you must call this routine at regular intervals to update the mouse state variables. To help you test your mouse polling code even if you are programming on a platform that doesn't require it, after the first time that you call this function Allegro will switch into polling mode, so from that point onwards you will have to call this routine in order to get any mouse input at all, regardless of whether the current driver actually needs to be polled or not. Returns zero on success, or a negative number on failure (ie. no mouse driver installed).

See also:

See Section 4.4 [mouse_needs_poll], page 43.

See Section 4.1 [install_mouse], page 41.

See Section 4.5 [mouse_x], page 43.

4.4 mouse_needs_poll

`int mouse_needs_poll();`

Returns TRUE if the current mouse driver is operating in polling mode.

See also:

See Section 4.3 [poll_mouse], page 42.

See Section 4.1 [install_mouse], page 41.

See Section 4.5 [mouse_x], page 43.

4.5 mouse_x

`extern volatile int mouse_x;`

`extern volatile int mouse_y;`

`extern volatile int mouse_z;`

`extern volatile int mouse_b;`

`extern volatile int mouse_pos;`

Global variables containing the current mouse position and button state. Wherever possible these values will be updated asynchronously, but if mouse_needs_poll() returns TRUE, you must manually call poll_mouse() to update them with the current input state. The mouse_x and mouse_y positions are integers ranging from zero to the bottom right corner of the screen. The mouse_z variable holds the current wheel position, when using an input driver that supports wheel mice. The mouse_b variable is a bitfield indicating the state of each button: bit 0 is the left button, bit 1 the right, and bit 2 the middle button. For example:

```
if (mouse_b & 1)
   printf("Left button is pressed\n");

if (!(mouse_b & 2))
   printf("Right button is not pressed\n");
```

The mouse_pos variable has the current X coordinate in the high word and the Y in the low word. This may be useful in tight polling loops where a mouse interrupt could occur between your reading of the two separate variables, since you can copy this value into a local variable with a single instruction and then split it up at your leisure.

See also:

See Section 4.1 [install_mouse], page 41.

See Section 4.3 [poll_mouse], page 42.

See Section 4.4 [mouse_needs_poll], page 43.

4.6 mouse_sprite

`extern BITMAP *mouse_sprite;`

`extern int mouse_x_focus;`

`extern int mouse_y_focus;`

Global variables containing the current mouse sprite and the focus point. These are read-only, and only to be modified using the set_mouse_sprite() and set_mouse_sprite_focus() functions.

See also:

See Section 4.16 [set_mouse_sprite], page 46.

See Section 4.17 [set_mouse_sprite_focus], page 47.

4.7 show_mouse

`void show_mouse(BITMAP *bmp);`

Tells Allegro to display a mouse pointer on the screen. This will only work if the timer module has been installed. The mouse pointer will be drawn onto the specified bitmap, which should normally be 'screen' (see later for information about bitmaps). To hide the mouse pointer, call show_mouse(NULL). Warning: if you draw anything onto the screen while the pointer is visible, a mouse movement interrupt could occur in the middle of your drawing operation. If this happens the mouse buffering and SVGA bank switching code will get confused and will leave 'mouse droppings' all over the screen. To prevent this, you must make sure you turn off the mouse pointer whenever you draw onto the screen.

See also:

See Section 4.1 [install_mouse], page 41.

See Section 5.1 [install_timer], page 48.

See Section 4.16 [set_mouse_sprite], page 46.

See Section 4.8 [scare_mouse], page 44.

See Section 4.11 [freeze_mouse_flag], page 45.

4.8 scare_mouse

`void scare_mouse();`

Helper for hiding the mouse pointer prior to a drawing operation. This will temporarily get rid of the pointer, but only if that is really required (ie. the mouse is visible, and is displayed on the physical screen rather than some other memory surface, and it is not a hardware cursor). The previous mouse state is stored for subsequent calls to unscare_mouse().

See also:

See Section 4.10 [unscare_mouse], page 45.

See Section 4.9 [scare_mouse_area], page 45.

See Section 4.7 [show_mouse], page 44.

4.9 scare_mouse_area

```
void scare_mouse_area(int x, int y, int w, int h);
```

Like scare_mouse(), but will only hide the cursor if it is inside the specified rectangle. Otherwise the cursor will simply be frozen in place until you call unscare_mouse(), so it cannot interfere with your drawing.

See also:

See Section 4.10 [unscare_mouse], page 45.

See Section 4.8 [scare_mouse], page 44.

See Section 4.7 [show_mouse], page 44.

4.10 unscare_mouse

```
void unscare_mouse();
```

Undoes the effect of a previous call to scare_mouse() or scare_mouse_area(), restoring the original pointer state.

See also:

See Section 4.8 [scare_mouse], page 44.

See Section 4.9 [scare_mouse_area], page 45.

4.11 freeze_mouse_flag

```
extern volatile int freeze_mouse_flag;
```

If this flag is set, the mouse pointer won't be redrawn when the mouse moves. This can avoid the need to hide the pointer every time you draw to the screen, as long as you make sure your drawing doesn't overlap with the current pointer position.

See also:

See Section 4.7 [show_mouse], page 44.

4.12 position_mouse

```
void position_mouse(int x, int y);
```

Moves the mouse to the specified screen position. It is safe to call even when a mouse pointer is being displayed.

See also:
See Section 4.1 [install_mouse], page 41.
See Section 4.13 [position_mouse_z], page 46.
See Section 4.14 [set_mouse_range], page 46.
See Section 4.15 [set_mouse_speed], page 46.

4.13 position_mouse_z

`void position_mouse_z(int z);`

Sets the mouse wheel position variable to the specified value.

See also:
See Section 4.1 [install_mouse], page 41.
See Section 4.12 [position_mouse], page 45.

4.14 set_mouse_range

`void set_mouse_range(int x1, int y1, int x2, int y2);`

Sets the area of the screen within which the mouse can move. Pass the top left corner and the bottom right corner (inclusive). If you don't call this function the range defaults to (0, 0, SCREEN_W-1, SCREEN_H-1).

See also:
See Section 4.1 [install_mouse], page 41.
See Section 4.15 [set_mouse_speed], page 46.
See Section 4.12 [position_mouse], page 45.

4.15 set_mouse_speed

`void set_mouse_speed(int xspeed, int yspeed);`

Sets the mouse speed. Larger values of xspeed and yspeed represent slower mouse movement: the default for both is 2.

See also:
See Section 4.1 [install_mouse], page 41.
See Section 4.14 [set_mouse_range], page 46.
See Section 4.12 [position_mouse], page 45.

4.16 set_mouse_sprite

```
void set_mouse_sprite(BITMAP *sprite);
```

You don't like my mouse pointer? No problem. Use this function to supply an alternative of your own. If you change the pointer and then want to get my lovely arrow back again, call set_mouse_sprite(NULL).

As a bonus, set_mouse_sprite(NULL) uses the current palette in choosing colors for the arrow. So if your arrow mouse sprite looks ugly after changing the palette, call set_mouse_sprite(NULL).

See also:

See Section 4.1 [install_mouse], page 41.

See Section 4.7 [show_mouse], page 44.

See Section 4.17 [set_mouse_sprite_focus], page 47.

4.17 set_mouse_sprite_focus

```
void set_mouse_sprite_focus(int x, int y);
```

The mouse focus is the bit of the pointer that represents the actual mouse position, ie. the (mouse_x, mouse_y) position. By default this is the top left corner of the arrow, but if you are using a different mouse pointer you might need to alter it.

See also:

See Section 4.16 [set_mouse_sprite], page 46.

4.18 get_mouse_mickeys

```
void get_mouse_mickeys(int *mickeyx, int *mickeyy);
```

Measures how far the mouse has moved since the last call to this function. The mouse will continue to generate movement mickeys even when it reaches the edge of the screen, so this form of input can be useful for games that require an infinite range of mouse movement.

See also:

See Section 4.1 [install_mouse], page 41.

4.19 mouse_callback

```
extern void (*mouse_callback)(int flags);
```

Called by the interrupt handler whenever the mouse moves or one of the buttons changes state. This function must be in locked memory, and must execute _very_ quickly! It is passed the event flags that triggered the call, which is a bitmask containing any of the values MOUSE_FLAG_MOVE, MOUSE_FLAG_LEFT_DOWN, MOUSE_FLAG_LEFT_UP, MOUSE_FLAG_RIGHT_DOWN, MOUSE_FLAG_RIGHT_UP, MOUSE_FLAG_MIDDLE_DOWN, MOUSE_FLAG_MIDDLE_UP, and MOUSE_FLAG_MOVE_Z.

See also:
See Section 4.1 [install_mouse], page 41.

5 Timer routines

Allegro can set up several virtual timer functions, all going at different speeds.

Under DOS it will constantly reprogram the clock to make sure they are all called at the correct times. Because they alter the low level timer chip settings, these routines should not be used together with other DOS timer functions like the djgpp uclock() routine. Moreover, the FPU state is not preserved across Allegro interrupts so you ought not to use floating point or MMX code inside timer interrupt handlers.

Under other platforms, they are usually implemented using threads, which run parallel to the main thread. Therefore timer callbacks on such platforms will not block the main thread when called, so you may need to use appropriate synchronisation devices (eg. mutexes, semaphores, etc.) when accessing data that is shared by a callback and the main thread. (Currently Allegro does not provide such devices.)

5.1 install_timer

`int install_timer();`

Installs the Allegro timer interrupt handler. You must do this before installing any user timer routines, and also before displaying a mouse pointer, playing FLI animations or MIDI music, and using any of the GUI routines. Returns zero on success, or a negative number on failure (but you may decide not to check the return value as this function is very unlikely to fail).

See also:

See Section 5.2 [remove_timer], page 48.

See Section 5.3 [install_int], page 49.

5.2 remove_timer

`void remove_timer();`

Removes the Allegro timer handler (and, under DOS, passes control of the clock back to the operating system). You don't normally need to bother calling this, because allegro_exit() will do it for you.

See also:

See Section 5.1 [install_timer], page 48.

See Section 1.3 [allegro_exit], page 1.

5.3 install_int

`int install_int(void (*proc)(), int speed);`

Installs a user timer handler, with the speed given as the number of milliseconds between ticks. This is the same thing as install_int_ex(proc, MSEC_TO_TIMER(speed)). If you call this routine without having first installed the timer module, install_timer() will be called automatically. If there is no room to add a new user timer, install_int() will return a negative number, otherwise it returns zero.

See also:
See Section 5.1 [install_timer], page 48.
See Section 5.5 [remove_int], page 50.
See Section 5.4 [install_int_ex], page 49.
See Section 5.6 [install_param_int], page 51.

5.4 install_int_ex

`int install_int_ex(void (*proc)(), int speed);`

Adds a function to the list of user timer handlers or, if it is already installed, retroactively adjusts its speed (i.e makes as though the speed change occured precisely at the last tick). The speed is given in hardware clock ticks, of which there are 1193181 a second. You can convert from other time formats to hardware clock ticks with the macros:

```
     SECS_TO_TIMER(secs)  - give the number of seconds between
                            each tick
     MSEC_TO_TIMER(msec)  - give the number of milliseconds
                            between ticks
     BPS_TO_TIMER(bps)    - give the number of ticks each second
     BPM_TO_TIMER(bpm)    - give the number of ticks per minute
```

If there is no room to add a new user timer, install_int_ex() will return a negative number, otherwise it returns zero. There can only be sixteen timers in use at a time, and some other parts of Allegro (the GUI code, the mouse pointer display routines, rest(), the FLI player, and the MIDI player) need to install handlers of their own, so you should avoid using too many at the same time. If you call this routine without having first installed the timer module, install_timer() will be called automatically.

Your function will be called by the Allegro interrupt handler and not directly by the processor, so it can be a normal C function and does not need a special wrapper. You should be aware, however, that it will be called in an interrupt context, which imposes a lot of restrictions on what you can do in it. It should not use large amounts of stack, it must not make any calls to the operating system, use C library functions, or contain any floating point code, and it must execute very quickly. Don't try to do lots of complicated code in a timer handler:

as a general rule you should just set some flags and respond to these later in your main control loop.

In a DOS protected mode environment like djgpp, memory is virtualised and can be swapped to disk. Due to the non-reentrancy of DOS, if a disk swap occurs inside an interrupt handler the system will die a painful death, so you need to make sure you lock all the memory (both code and data) that is touched inside timer routines. Allegro will lock everything it uses, but you are responsible for locking your handler functions. The macros LOCK_VARIABLE (variable), END_OF_FUNCTION (function_name), END_OF_STATIC_FUNCTION (function_name), and LOCK_FUNCTION (function_name) can be used to simplify this task. For example, if you want an interrupt handler that increments a counter variable, you should write:

```
volatile int counter;

void my_timer_handler()
{
   counter++;
}

END_OF_FUNCTION(my_timer_handler)
```

and in your initialisation code you should lock the memory:

```
LOCK_VARIABLE(counter);
LOCK_FUNCTION(my_timer_handler);
```

Obviously this can get awkward if you use complicated data structures and call other functions from within your handler, so you should try to keep your interrupt routines as simple as possible.

See also:

See Section 5.1 [install_timer], page 48.

See Section 5.5 [remove_int], page 50.

See Section 5.3 [install_int], page 49.

See Section 5.7 [install_param_int_ex], page 51.

5.5 remove_int

`void remove_int(void (*proc)());`

Removes a function from the list of user interrupt routines. At program termination, allegro_exit() does this automatically.

See also:

See Section 5.3 [install_int], page 49.

See Section 5.4 [install_int_ex], page 49.

See Section 5.8 [remove_param_int], page 51.

5.6 install_param_int

`int install_param_int(void (*proc)(void *), void *param, int speed);`

Like install_int(), but the callback routine will be passed a copy of the specified void pointer parameter. To disable the handler, use remove_param_int() instead of remove_int().

See also:

See Section 5.1 [install_timer], page 48.

See Section 5.8 [remove_param_int], page 51.

See Section 5.7 [install_param_int_ex], page 51.

See Section 5.3 [install_int], page 49.

5.7 install_param_int_ex

`int install_param_int_ex(void (*proc)(void *), void *param, int speed);`

Like install_int_ex(), but the callback routine will be passed a copy of the specified void pointer parameter. To disable the handler, use remove_param_int() instead of remove_int().

See also:

See Section 5.1 [install_timer], page 48.

See Section 5.8 [remove_param_int], page 51.

See Section 5.6 [install_param_int], page 51.

See Section 5.4 [install_int_ex], page 49.

5.8 remove_param_int

`void remove_param_int(void (*proc)(void *), void *param);`

Like remove_int(), but for use with timer callbacks that have parameter values. If there is more than one copy of the same callback active at a time, it identifies which one to remove by checking the parameter value (so you can't have more than one copy of a handler using an identical parameter).

See also:

See Section 5.6 [install_param_int], page 51.

See Section 5.7 [install_param_int_ex], page 51.

See Section 5.5 [remove_int], page 50.

5.9 timer_can_simulate_retrace

`int timer_can_simulate_retrace()`

Checks whether it is possible to sync the timer module with the monitor retrace, given the current platform and environment (at the moment this is only possible when running in clean DOS mode in a VGA or mode-X resolution). Returns non-zero if simulation is possible.

See also:

See Section 5.10 [timer_simulate_retrace], page 52.

See Section 5.11 [timer_is_using_retrace], page 53.

5.10 timer_simulate_retrace

`void timer_simulate_retrace(int enable);`

The DOS timer handler can be used to simulate vertical retrace interrupts. A retrace interrupt can be extremely useful for implementing smooth animation, but unfortunately the VGA hardware doesn't support it. The EGA did, and some SVGA chipsets do, but not enough, and not in a sufficiently standardised way, for it to be useful. Allegro works around this by programming the timer to generate an interrupt when it thinks a retrace is next likely to occur, and polling the VGA inside the interrupt handler to make sure it stays in sync with the monitor refresh. This works quite well in some situations, but there are a lot of caveats:

- You can't use the retrace simulator in SVGA modes. It will work with some chipsets, but not others, and it conflicts with most VESA implementations. Retrace simulation is only reliable in VGA mode 13h and mode-X.

- Retrace simulation doesn't work under win95, because win95 returns garbage when I try to read the elapsed time from the PIT. If anyone knows how I can make this work, please tell me!

- Retrace simulation involves a lot of waiting around in the timer handler with interrupts disabled. This will significantly slow down your entire system, and may also cause static when playing samples on SB 1.0 cards (because they don't support auto-initialised DMA: SB 2.0 and above will be fine).

Bearing all those problems in mind, I'd strongly advise against relying on the retrace simulator. If you are coding in mode-X, and don't care about your program working under win95, it is great, but it would be a good idea to give the user an option to disable it.

Retrace simulation must be enabled before you use the triple buffering functions in a mode-X resolution. It can also be useful for simple retrace detection, because the polling vsync() function can occasionally miss retraces if a soundcard or timer interrupt occurs at exactly the same time as the retrace. When retrace interrupt simulation is enabled, vsync() will check the retrace_count variable rather than polling the VGA, so it won't miss retraces even if they are masked by other interrupts.

See also:
See Section 5.1 [install_timer], page 48.
See Section 5.12 [retrace_count], page 53.
See Section 5.13 [retrace_proc], page 53.
See Section 8.14 [request_scroll], page 78.
See Section 11.1 [vsync], page 93.
See Section 5.9 [timer_can_simulate_retrace], page 52.
See Section 5.11 [timer_is_using_retrace], page 53.
See Section 8.12 [enable_triple_buffer], page 77.

5.11 timer_is_using_retrace

`int timer_is_using_retrace()`

Checks whether the timer module is currently synced with the monitor retrace or not. Returns non-zero if it is.

See also:
See Section 5.10 [timer_simulate_retrace], page 52.
See Section 5.9 [timer_can_simulate_retrace], page 52.

5.12 retrace_count

`extern volatile int retrace_count;`

If the retrace simulator is installed, this is incremented on each vertical retrace, otherwise it is incremented 70 times a second (ignoring retraces). This provides a useful way of controlling the speed of your program without the hassle of installing user timer functions.

The speed of retraces varies depending on the graphics mode. In mode 13h and 200/400 line mode-X resolutions there are 70 retraces a second, and in 240/480 line modes there are 60. It can be as low as 50 (in 376x282 mode) or as high as 92 (in 400x300 mode).

See also:
See Section 5.10 [timer_simulate_retrace], page 52.
See Section 5.11 [timer_is_using_retrace], page 53.
See Section 5.13 [retrace_proc], page 53.

5.13 retrace_proc

`extern void (*retrace_proc)();`

If the retrace simulator is installed, this function is called during every vertical retrace, otherwise it is called 70 times a second (ignoring retraces). Set it to

NULL to disable the callback. The function must obey the same rules as regular timer handlers (ie. it must be locked, and mustn't call OS or libc functions) but even more so: it must execute _very_ quickly, or it will mess up the timer synchronisation. The only use I can see for this function is for doing palette manipulations, because triple buffering can be done with the request_scroll() function, and the retrace_count variable can be used for timing your code. If you want to alter the palette in the retrace_proc you should use the inline _set_color() function rather than the regular set_color() or set_palette(), and you shouldn't try to alter more than two or three palette entries in a single retrace.

See also:

See Section 5.10 [timer_simulate_retrace], page 52.

See Section 5.11 [timer_is_using_retrace], page 53.

See Section 5.12 [retrace_count], page 53.

See Section 11.3 [_set_color], page 93.

5.14 rest

`void rest(long time);`

Once Allegro has taken over the timer the standard delay() function will no longer work, so you should use this routine instead. The time is given in milliseconds.

See also:

See Section 5.1 [install_timer], page 48.

See Section 5.15 [rest_callback], page 54.

5.15 rest_callback

`void rest_callback(long time, void (*callback)())`

Like rest(), but continually calls the specified function while it is waiting for the required time to elapse.

See also:
See Section 5.1 [install_timer], page 48.
See Section 5.14 [rest], page 54.

6 Keyboard routines

The Allegro keyboard handler provides both buffered input and a set of flags storing the current state of each key. Note that it is not possible to correctly detect every combination of keys, due to the design of the PC keyboard. Up to two or three keys at a time will work fine, but if you press more than that the extras are likely to be ignored (exactly which combinations are possible seems to vary from one keyboard to another).

6.1 install_keyboard

`int install_keyboard();`

Installs the Allegro keyboard interrupt handler. You must call this before using any of the keyboard input routines. Once you have set up the Allegro handler, you can no longer use operating system calls or C library functions to access the keyboard. Returns zero on success, or a negative number on failure (but you may decide not to check the return value as this function is very unlikely to fail). Note that on some platforms the keyboard won't work unless you have set a graphic mode, even if this function returns zero before calling set_gfx_mode.

See also:

See Section 6.2 [remove_keyboard], page 55.

See Section 6.4 [poll_keyboard], page 56.

See Section 6.6 [key], page 57.

See Section 6.8 [keypressed], page 58.

See Section 6.9 [readkey], page 59.

See Section 6.10 [ureadkey], page 59.

See Section 6.14 [keyboard_callback], page 60.

See Section 6.15 [keyboard_ucallback], page 61.

See Section 6.16 [keyboard_lowlevel_callback], page 61.

See Section 6.20 [three_finger_flag], page 62.

See Section 6.21 [key_led_flag], page 63.

See Section 6.17 [set_leds], page 62.

See Section 6.18 [set_keyboard_rate], page 62.

See Section 8.6 [set_gfx_mode], page 70.

See Section 3.23 [standard config variables], page 36.

6.2 remove_keyboard

`void remove_keyboard();`

Removes the keyboard handler, returning control to the operating system. You don't normally need to bother calling this, because allegro_exit() will do it for you.

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 1.3 [allegro_exit], page 1.

6.3 install_keyboard_hooks

`void install_keyboard_hooks(int (*keypressed)(), int (*readkey)());`

You should only use this function if you *aren't* using the rest of the keyboard handler. It should be called in the place of install_keyboard(), and lets you provide callback routines to detect and read keypresses, which will be used by the main keypressed() and readkey() functions. This can be useful if you want to use Allegro's GUI code with a custom keyboard handler, as it provides a way for the GUI to get keyboard input from your own code, bypassing the normal Allegro input system.

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.8 [keypressed], page 58.

See Section 6.9 [readkey], page 59.

6.4 poll_keyboard

`int poll_keyboard();`

Wherever possible, Allegro will read the keyboard input asynchronously (ie. from inside an interrupt handler), but on some platforms that may not be possible, in which case you must call this routine at regular intervals to update the keyboard state variables. To help you test your keyboard polling code even if you are programming on a platform that doesn't require it, after the first time that you call this function Allegro will switch into polling mode, so from that point onwards you will have to call this routine in order to get any keyboard input at all, regardless of whether the current driver actually needs to be polled or not. The keypressed(), readkey(), and ureadkey() functions call poll_keyboard() automatically, so you only need to use this function when accessing the key[] array and key_shifts variable. Returns zero on success, or a negative number on failure (ie. no keyboard driver installed).

See also:

See Section 6.5 [keyboard_needs_poll], page 56.

See Section 6.1 [install_keyboard], page 55.

See Section 6.6 [key], page 57.

See Section 6.7 [key_shifts], page 58.

6.5 keyboard_needs_poll

`int keyboard_needs_poll();`

Returns TRUE if the current keyboard driver is operating in polling mode.

See also:

See Section 6.4 [poll_keyboard], page 56.

See Section 6.1 [install_keyboard], page 55.

See Section 6.6 [key], page 57.

6.6 key

`extern volatile char key[KEY_MAX];`

Array of flags indicating the state of each key, ordered by scancode. Wherever possible these values will be updated asynchronously, but if keyboard_needs_poll() returns TRUE, you must manually call poll_keyboard() to update them with the current input state. The scancodes are defined in allegro/keyboard.h as a series of KEY_* constants (and are also listed below). For example, you could write:

```
if (key[KEY_SPACE])
   printf("Space is pressed\n");
```

Note that the array is supposed to represent which keys are physically held down and which keys are not, so it is semantically read-only.

These are the keyboard scancodes:

```
KEY_A ... KEY_Z,
KEY_0 ... KEY_9,
KEY_0_PAD ... KEY_9_PAD,
KEY_F1 ... KEY_F12,

KEY_ESC, KEY_TILDE, KEY_MINUS, KEY_EQUALS,
KEY_BACKSPACE, KEY_TAB, KEY_OPENBRACE, KEY_CLOSEBRACE,
KEY_ENTER, KEY_COLON, KEY_QUOTE, KEY_BACKSLASH,
KEY_BACKSLASH2, KEY_COMMA, KEY_STOP, KEY_SLASH,
KEY_SPACE,

KEY_INSERT, KEY_DEL, KEY_HOME, KEY_END, KEY_PGUP,
KEY_PGDN, KEY_LEFT, KEY_RIGHT, KEY_UP, KEY_DOWN,

KEY_SLASH_PAD, KEY_ASTERISK, KEY_MINUS_PAD,
KEY_PLUS_PAD, KEY_DEL_PAD, KEY_ENTER_PAD,

KEY_PRTSCR, KEY_PAUSE,

KEY_ABNT_C1, KEY_YEN, KEY_KANA, KEY_CONVERT, KEY_NOCONVERT,
KEY_AT, KEY_CIRCUMFLEX, KEY_COLON2, KEY_KANJI,

KEY_LSHIFT, KEY_RSHIFT,
KEY_LCONTROL, KEY_RCONTROL,
KEY_ALT, KEY_ALTGR,
KEY_LWIN, KEY_RWIN, KEY_MENU,
KEY_SCRLOCK, KEY_NUMLOCK, KEY_CAPSLOCK
```

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.4 [poll_keyboard], page 56.

See Section 6.7 [key_shifts], page 58.

6.7 key_shifts

`extern volatile int key_shifts;`

Bitmask containing the current state of shift/ctrl/alt, the special Windows keys, and the accent escape characters. Wherever possible this value will be updated asynchronously, but if keyboard_needs_poll() returns TRUE, you must manually call poll_keyboard() to update it with the current input state. This can contain any of the flags:

KB_SHIFT_FLAG
KB_CTRL_FLAG
KB_ALT_FLAG
KB_LWIN_FLAG
KB_RWIN_FLAG
KB_MENU_FLAG
KB_SCROLOCK_FLAG
KB_NUMLOCK_FLAG
KB_CAPSLOCK_FLAG
KB_INALTSEQ_FLAG
KB_ACCENT1_FLAG
KB_ACCENT2_FLAG
KB_ACCENT3_FLAG
KB_ACCENT4_FLAG

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.4 [poll_keyboard], page 56.

See Section 6.6 [key], page 57.

6.8 keypressed

`int keypressed();`

Returns TRUE if there are keypresses waiting in the input buffer. This is equivalent to the libc kbhit() function.

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.9 [readkey], page 59.

See Section 6.10 [ureadkey], page 59.
See Section 6.19 [clear_keybuf], page 62.
See Section 6.12 [simulate_keypress], page 60.
See Section 6.13 [simulate_ukeypress], page 60.

6.9 readkey

`int readkey();`

Returns the next character from the keyboard buffer, in ASCII format. If the buffer is empty, it waits until a key is pressed. The low byte of the return value contains the ASCII code of the key, and the high byte the scancode. The scancode remains the same whatever the state of the shift, ctrl and alt keys, while the ASCII code is affected by shift and ctrl in the normal way (shift changes case, ctrl+letter gives the position of that letter in the alphabet, eg. ctrl+A = 1, ctrl+B = 2, etc). Pressing alt+key returns only the scancode, with a zero ASCII code in the low byte. For example:

```
if ((readkey() & 0xff) == 'd')         // by ASCII code
   printf("You pressed 'd'\n");

if ((readkey() >> 8) == KEY_SPACE)     // by scancode
   printf("You pressed Space\n");

if ((readkey() & 0xff) == 3)           // ctrl+letter
   printf("You pressed Control+C\n");

if (readkey() == (KEY_X << 8))         // alt+letter
   printf("You pressed Alt+X\n");
```

This function cannot return character values greater than 255. If you need to read Unicode input, use ureadkey() instead.

See also:
See Section 6.1 [install_keyboard], page 55.
See Section 6.10 [ureadkey], page 59.
See Section 6.8 [keypressed], page 58.
See Section 6.19 [clear_keybuf], page 62.
See Section 6.12 [simulate_keypress], page 60.

6.10 ureadkey

`int ureadkey(int *scancode);`

Returns the next character from the keyboard buffer, in Unicode format. If the buffer is empty, it waits until a key is pressed. The return value contains the Unicode value of the key, and if not NULL, the pointer argument will be set to the scancode. Unlike readkey(), this function is able to return character values greater than 255.

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.9 [readkey], page 59.

See Section 6.8 [keypressed], page 58.

See Section 6.19 [clear_keybuf], page 62.

See Section 6.13 [simulate_ukeypress], page 60.

6.11 scancode_to_ascii

```
int scancode_to_ascii(int scancode);
```

Converts the given scancode to an ASCII character for that key, returning the unshifted uncapslocked result of pressing the key, or zero if the key isn't a character-generating key or the lookup can't be done.

6.12 simulate_keypress

```
void simulate_keypress(int key);
```

Stuffs a key into the keyboard buffer, just as if the user had pressed it. The parameter is in the same format returned by readkey().

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.13 [simulate_ukeypress], page 60.

See Section 6.8 [keypressed], page 58.

See Section 6.9 [readkey], page 59.

6.13 simulate_ukeypress

```
void simulate_ukeypress(int key, int scancode);
```

Stuffs a key into the keyboard buffer, just as if the user had pressed it. The parameter is in the same format returned by ureadkey().

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.12 [simulate_keypress], page 60.

See Section 6.8 [keypressed], page 58.

See Section 6.10 [ureadkey], page 59.

6.14 keyboard_callback

`extern int (*keyboard_callback)(int key);`

If set, this function is called by the keyboard handler in response to every keypress. It is passed a copy of the value that is about to be added into the input buffer, and can either return this value unchanged, return zero to cause the key to be ignored, or return a modified value to change what readkey() will later return. This routine executes in an interrupt context, so it must be in locked memory.

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.9 [readkey], page 59.

See Section 6.10 [ureadkey], page 59.

See Section 6.15 [keyboard_ucallback], page 61.

See Section 6.16 [keyboard_lowlevel_callback], page 61.

6.15 keyboard_ucallback

`extern int (*keyboard_ucallback)(int key, int *scancode);`

Unicode-aware version of keyboard_callback(). If set, this function is called by the keyboard handler in response to every keypress. It is passed the character value and scancode that are about to be added into the input buffer, can modify the scancode value, and returns a new or modified key code. If it both sets the scancode to zero and returns zero, the keypress will be ignored. This routine executes in an interrupt context, so it must be in locked memory.

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.9 [readkey], page 59.

See Section 6.10 [ureadkey], page 59.

See Section 6.14 [keyboard_callback], page 60.

See Section 6.16 [keyboard_lowlevel_callback], page 61.

6.16 keyboard_lowlevel_callback

`extern void (*keyboard_lowlevel_callback)(int scancode);`

If set, this function is called by the keyboard handler in response to every keyboard event, both presses and releases. It will be passed a raw keyboard scancode byte, with the top bit clear if the key has been pressed or set if it was released. This routine executes in an interrupt context, so it must be in locked memory.

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.14 [keyboard_callback], page 60.

See Section 6.15 [keyboard_ucallback], page 61.

6.17 set_leds

`void set_leds(int leds);`

Overrides the state of the keyboard LED indicators. The parameter is a bitmask containing any of the values KB_SCROLOCK_FLAG, KB_NUMLOCK_FLAG, and KB_CAPSLOCK_FLAG, or -1 to restore the default behavior.

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.21 [key_led_flag], page 63.

6.18 set_keyboard_rate

`void set_keyboard_rate(int delay, int repeat);`

Sets the keyboard repeat rate. Times are given in milliseconds. Passing zero times will disable the key repeat.

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.9 [readkey], page 59.

See Section 6.10 [ureadkey], page 59.

6.19 clear_keybuf

`void clear_keybuf();`

Empties the keyboard buffer.

See also:

See Section 6.1 [install_keyboard], page 55.

See Section 6.8 [keypressed], page 58.

See Section 6.9 [readkey], page 59.

See Section 6.10 [ureadkey], page 59.

6.20 three_finger_flag

`extern int three_finger_flag;`

The djgpp keyboard handler provides an 'emergency exit' sequence which you can use to kill off your program. If you are running under DOS this is the three finger salute, ctrl+alt+del. Most multitasking OS's will trap this combination before it reaches the Allegro handler, in which case you can use the alternative ctrl+alt+end. If you want to disable this behaviour in release versions of your program, set this flag to FALSE.

See also:
See Section 6.1 [install_keyboard], page 55.

6.21 key_led_flag

```
extern int key_led_flag;
```

By default, the capslock, numlock, and scroll-lock keys toggle the keyboard LED indicators when they are pressed. If you are using these keys for input in your game (eg. capslock to fire) this may not be desirable, so you can clear this flag to prevent the LED's being updated.

See also:
See Section 6.1 [install_keyboard], page 55.
See Section 6.17 [set_leds], page 62.

7 Joystick routines

7.1 install_joystick

```
int install_joystick(int type);
```

Initialises the joystick, and calibrates the centre position value. The type parameter should usually be JOY_TYPE_AUTODETECT, or see the platform specific documentation for a list of the available drivers. You must call this routine before using any other joystick functions, and you should make sure that the joystick is in the middle position at the time. Returns zero on success. As soon as you have installed the joystick module, you will be able to read the button state and digital (on/off toggle) direction information, which may be enough for some games. If you want to get full analogue input, though, you need to use the calibrate_joystick() functions to measure the exact range of the inputs: see below.

See also:
See Section 7.2 [remove_joystick], page 64.
See Section 7.9 [load_joystick_data], page 68.
See Section 7.7 [calibrate_joystick], page 67.
See Section 7.6 [calibrate_joystick_name], page 67.
See Section 7.3 [poll_joystick], page 64.
See Section 3.23 [standard config variables], page 36.
See Section 34.1 [JOY_TYPE_*/DOS], page 227.

7.2 remove_joystick

`void remove_joystick();`

Removes the joystick handler. You don't normally need to bother calling this, because allegro_exit() will do it for you.

See also:

See Section 7.1 [install_joystick], page 63.

See Section 1.3 [allegro_exit], page 1.

7.3 poll_joystick

`int poll_joystick();`

The joystick is not interrupt driven, so you need to call this function every now and again to update the global position values. Returns zero on success or a negative number on failure (usually because no joystick driver was installed).

See also:

See Section 7.1 [install_joystick], page 63.

See Section 7.5 [joy], page 64.

See Section 7.4 [num_joysticks], page 64.

7.4 num_joysticks

`extern int num_joysticks;`

Global variable containing the number of active joystick devices. The current drivers support a maximum of four controllers.

See also:

See Section 7.1 [install_joystick], page 63.

See Section 7.5 [joy], page 64.

7.5 joy

`extern JOYSTICK_INFO joy[n];`

Global array of joystick state information, which is updated by the poll_joystick() function. Only the first num_joysticks elements will contain meaningful information. The JOYSTICK_INFO structure is defined as:

```
typedef struct JOYSTICK_INFO
{
   int flags;                         - status flags for this
                                        joystick
   int num_sticks;                    - how many stick inputs?
```

```
      int num_buttons;                  - how many buttons?
      JOYSTICK_STICK_INFO stick[n];     - stick state information
      JOYSTICK_BUTTON_INFO button[n];   - button state information
   } JOYSTICK_INFO;
```

The button status is stored in the structure:

```
   typedef struct JOYSTICK_BUTTON_INFO
   {
      int b;                            - boolean on/off flag
      char *name;                       - description of this
                                          button
   } JOYSTICK_BUTTON_INFO;
```

You may wish to display the button names as part of an input configuration screen to let the user choose what game function will be performed by each button, but in simpler situations you can safely assume that the first two elements in the button array will always be the main trigger controls.

Each joystick will provide one or more stick inputs, of varying types. These can be digital controls which snap to specific positions (eg. a gamepad controller, the coolie hat on a Flightstick Pro or Wingman Extreme, or a normal joystick which hasn't yet been calibrated), or they can be full analogue inputs with a smooth range of motion. Sticks may also have different numbers of axis, for example a normal directional control has two, but the Flightstick Pro throttle is only a single axis, and it is possible that the system could be extended in the future to support full 3d controllers. A stick input is described by the structure:

```
   typedef struct JOYSTICK_STICK_INFO
   {
      int flags;                        - status flags for this
                                          input
      int num_axis;                     - how many axis do we
                                          have?
      JOYSTICK_AXIS_INFO axis[n];       - axis state information
      char *name;                       - description of this
                                          input
   } JOYSTICK_STICK_INFO;
```

A single joystick may provide several different stick inputs, but you can safely assume that the first element in the stick array will always be the main directional controller.

Information about each of the stick axis is stored in the substructure:

```
   typedef struct JOYSTICK_AXIS_INFO
   {
      int pos;                          - analogue axis position
      int d1, d2;                       - digital axis position
      char *name;                       - description of this axis
```

```
     } JOYSTICK_AXIS_INFO;
```

This provides both analogue input in the pos field (ranging from -128 to 128 or from 0 to 255, depending on the type of the control), and digital values in the d1 and d2 fields. For example, when describing the X-axis position, the pos field will hold the horizontal position of the joystick, d1 will be set if it is moved left, and d2 will be set if it is moved right. Allegro will fill in all these values regardless of whether it is using a digital or analogue joystick, emulating the pos field for digital inputs by snapping it to the min, middle, and maximum positions, and emulating the d1 and d2 values for an analogue stick by comparing the current position with the centre point.

The joystick flags field may contain any combination of the bit flags:

JOYFLAG_DIGITAL
This control is currently providing digital input.

JOYFLAG_ANALOGUE
This control is currently providing analogue input.

JOYFLAG_CALIB_DIGITAL
This control will be capable of providing digital input once it has been calibrated, but is not doing this at the moment.

JOYFLAG_CALIB_ANALOGUE
This control will be capable of providing analogue input once it has been calibrated, but is not doing this at the moment.

JOYFLAG_CALIBRATE
Indicates that this control needs to be calibrated. Many devices require multiple calibration steps, so you should call the calibrate_joystick() function from a loop until this flag is cleared.

JOYFLAG_SIGNED
Indicates that the analogue axis position is in signed format, ranging from -128 to 128. This is the case for all 2d directional controls.

JOYFLAG_UNSIGNED
Indicates that the analogue axis position is in unsigned format, ranging from 0 to 255. This is the case for all 1d throttle controls.

Note for people who spell funny: in case you don't like having to type "analogue", there are some #define aliases in allegro/joystick.h that will allow you to write "analog" instead.

See also:
See Section 7.1 [install_joystick], page 63.
See Section 7.3 [poll_joystick], page 64.
See Section 7.4 [num_joysticks], page 64.
See Section 7.7 [calibrate_joystick], page 67.
See Section 7.6 [calibrate_joystick_name], page 67.

7.6 calibrate_joystick_name

`const char *calibrate_joystick_name(int n);`

Returns a text description for the next type of calibration that will be done on the specified joystick, or NULL if no more calibration is required.

See also:
See Section 7.1 [install_joystick], page 63.
See Section 7.7 [calibrate_joystick], page 67.
See Section 7.5 [joy], page 64.
See Section 7.4 [num_joysticks], page 64.

7.7 calibrate_joystick

`int calibrate_joystick(int n);`

Most joysticks need to be calibrated before they can provide full analogue input. This function performs the next operation in the calibration series for the specified stick, assuming that the joystick has been positioned in the manner described by a previous call to calibrate_joystick_name(), returning zero on success. For example, a simple routine to fully calibrate all the joysticks might look like:

```
int i;

for (i=0; i<;num_joysticks; i++) {
   while (joy[i].flags & JOYFLAG_CALIBRATE) {
      char *msg = calibrate_joystick_name(i);
      printf("%s, and press a key\n", msg);
      readkey();
      if (calibrate_joystick(i) != 0) {
         printf("oops!\n");
         exit(1);
      }
   }
}
```

See also:
See Section 7.1 [install_joystick], page 63.
See Section 7.6 [calibrate_joystick_name], page 67.
See Section 7.5 [joy], page 64.
See Section 7.4 [num_joysticks], page 64.

7.8 save_joystick_data

`int save_joystick_data(const char *filename);`

After all the headache of calibrating the joystick, you may not want to make your poor users repeat the process every time they run your program. Call this function to save the joystick calibration data into the specified configuration file, from which it can later be read by load_joystick_data(). Pass a NULL filename to write the data to the currently selected configuration file. Returns zero on success.

See also:

See Section 7.9 [load_joystick_data], page 68.

See Section 3.1 [set_config_file], page 28.

7.9 load_joystick_data

`int load_joystick_data(const char *filename);`

Restores calibration data previously stored by save_joystick_data() or the setup utility. This sets up all aspects of the joystick code: you don't even need to call install_joystick() if you are using this function. Pass a NULL filename to read the data from the currently selected configuration file. Returns zero on success: if it fails the joystick state is undefined and you must reinitialise it from scratch.

See also:

See Section 7.1 [install_joystick], page 63.

See Section 7.8 [save_joystick_data], page 67.

See Section 3.1 [set_config_file], page 28.

7.10 initialise_joystick

`int initialise_joystick();`

Deprecated. Use install_joystick() instead.

See also:

See Section 7.1 [install_joystick], page 63.

8 Graphics modes

8.1 set_color_depth

`void set_color_depth(int depth);`

Sets the pixel format to be used by subsequent calls to set_gfx_mode() and create_bitmap(). Valid depths are 8 (the default), 15, 16, 24, and 32 bits.

Note that you can retrieve the pixel format currently in use by calling bitmap_color_depth() on the 'screen' bitmap, once a graphics mode has been set.

See also:

See Section 8.6 [set_gfx_mode], page 70.

See Section 10.11 [set_color_conversion], page 90.

See Section 12.3 [makecol], page 100.

See Section 12.9 [getr], page 102.

See Section 9.9 [bitmap_color_depth], page 83.

8.2 request_refresh_rate

`void request_refresh_rate(int rate);`

Requests that the next call to set_gfx_mode() try to use the specified refresh rate, if possible. Not all drivers are able to control this at all, and even when they can, not all rates will be possible on all hardware, so the actual settings may differ from what you requested. After you call set_gfx_mode(), you can use get_refresh_rate() to find out what was actually selected. At the moment only the DOS VESA 3.0, X DGA 2.0 and some Windows DirectX drivers support this function. The speed is specified in Hz, eg. 60, 70. To return to the normal default selection, pass a rate value of zero.

See also:

See Section 8.6 [set_gfx_mode], page 70.

See Section 8.3 [get_refresh_rate], page 69.

8.3 get_refresh_rate

`int get_refresh_rate(void);`

Returns the current refresh rate, if known (not all drivers are able to report this information). Returns zero if the actual rate is unknown.

See also:

See Section 8.2 [request_refresh_rate], page 69.

8.4 get_gfx_mode_list

`GFX_MODE_LIST *get_gfx_mode_list(int card);`

Attempts to create a list of all the supported video modes for a certain graphics driver. This function returns a pointer to a list structure of the type GFX_MODE_LIST which has the following definition:

```
typedef struct GFX_MODE_LIST {
   int num_modes;
```

```
      GFX_MODE *mode;
   } GFX_MODE_LIST;
```

If this function returns NULL, it means the call failed. The mode entry points to the actual list of video modes.

```
   typedef struct GFX_MODE {
      int width, height, bpp;
   } GFX_MODE;
```

This list is terminated with an { 0, 0, 0 } entry.

Note that the card parameter must refer to a _real_ driver. This function fails if you pass GFX_SAFE, GFX_AUTODETECT, or any other "magic" driver.

See also:

See Section 8.5 [destroy_gfx_mode_list], page 70.

See Section 8.6 [set_gfx_mode], page 70.

See Section 8.1 [set_color_depth], page 68.

8.5 destroy_gfx_mode_list

`void destroy_gfx_mode_list(GFX_MODE_LIST *mode_list);`

Removes the mode list created by get_gfx_mode_list() from memory.

See also:

See Section 8.4 [get_gfx_mode_list], page 69.

See Section 8.6 [set_gfx_mode], page 70.

See Section 8.1 [set_color_depth], page 68.

8.6 set_gfx_mode

`int set_gfx_mode(int card, int w, int h, int v_w, int v_h);`

Switches into graphics mode. The card parameter should usually be GFX_AUTODETECT, GFX_AUTODETECT_FULLSCREEN or GFX_AUTODETECT_WINDOWED, or see the platform specific documentation for a list of the available drivers. The w and h parameters specify what screen resolution you want.

The v_w and v_h parameters specify the minimum virtual screen size, in case you need a large virtual screen for hardware scrolling or page flipping. You should set them to zero if you don't care about the virtual screen size. Virtual screens can cause a lot of confusion, but they are really quite simple. Warning: patronising explanation coming up, so you may wish to skip the rest of this paragraph :-) Think of video memory as a rectangular piece of paper which is being viewed through a small hole (your monitor) in a bit of cardboard. Since the paper is bigger than the hole you can only see part of it at any one time, but by sliding the cardboard around you can alter which portion of the image is visible. You could just leave the hole in one position and ignore the parts of video memory that aren't visible, but you can get all sorts of useful effects by sliding the screen window around, or by drawing images in a hidden part of video memory and then flipping across to display them.

For example, you could select a 640x480 mode in which the monitor acts as a window onto a 1024x1024 virtual screen, and then move the visible screen around in this larger area. Initially, with the visible screen positioned at the top left corner of video memory, this setup would look like:

```
(0,0)------------(640,0)----(1024,0)
  |                  |           |
  |  visible screen  |           |
  |                  |           |
(0,480)----------(640,480)       |
  |                              |
  |   the rest of video memory   |
  |                              |
(0,1024)--------------------(1024,1024)
```

What's that? You are viewing this with a proportional font? Hehehe.

When you call set_gfx_mode(), the v_w and v_h parameters represent the minimum size of virtual screen that is acceptable for your program. The range of possible sizes is usually very restricted, and Allegro is likely to end up creating a virtual screen much larger than the one you request. On an SVGA card with one megabyte of vram you can count on getting a 1024x1024 virtual screen (256 colors) or 1024x512 (15 or 16 bpp), and with 512k vram you can get 1024x512 (256 color). Other sizes may or may not be possible: don't assume that they will work. In mode-X the virtual width can be any multiple of eight greater than or equal to the physical screen width, and the virtual height will be set accordingly (the VGA has 256k of vram, so the virtual height will be 256*1024/virtual_width).

After you select a graphics mode, the physical and virtual screen sizes can be checked with the macros SCREEN_W, SCREEN_H, VIRTUAL_W, and VIRTUAL_H.

If Allegro is unable to select an appropriate mode, set_gfx_mode() returns a negative number and stores a description of the problem in allegro_error. Otherwise it returns zero.

As a special case, if you use the magic driver code GFX_SAFE, Allegro will guarantee that the mode will always be set correctly. It will try to select the resolution that you request, and if that fails, it will fall back upon whatever mode is known to be reliable on the current platform (this is 320x200 VGA mode under DOS, a 640x480 resolution under Windows, the actual framebuffer's resolution under Linux if it's supported, etc). If it absolutely cannot set any graphics mode at all, it will return negative as usual, meaning that there's no possible video output on the machine, and that you should abort your program immediately, possibly after notifying this to the user with allegro_message. This fake driver is useful for situations where you just want to get into some kind of workable display mode, and can't be bothered with trying multiple different resolutions and doing all the error checking yourself. Note however, that after a successful call to set_gfx_mode with this driver, you cannot make any assumptions about the width, height or color depth of the screen: your code will have to deal with this little detail.

Finally, GFX_TEXT is another magic driver which usually closes any previously opened graphic mode, making you unable to use the global variable screen, and in those environments that have text modes, sets one previously used or the closest match to that (usually 80x25). With this driver the size parameters of set_gfx_mode don't mean anything, so you can leave them all to zero or any other number you prefer.

See also:

See Section 8.1 [set_color_depth], page 68.

See Section 8.2 [request_refresh_rate], page 69.

See Section 9.1 [screen], page 80.

See Section 8.11 [gfx_capabilities], page 74.

See Section 1.6 [allegro_error], page 2.

See Section 3.23 [standard config variables], page 36.

See Section 34.2 [GFX_*/DOS], page 230.

See Section 35.1 [GFX_*/Windows], page 235.

See Section 36.2 [GFX_*/X], page 243.

See Section 36.1 [GFX_*/Linux], page 242.

See Section 37.1 [GFX_*/BeOS], page 245.

8.7 set_display_switch_mode

```
int set_display_switch_mode(int mode);
```

Sets how the program should handle being switched into the background, if the user tabs away from it. Not all of the possible modes will be supported by every graphics driver on every platform: you must call this routine after initializing the graphics driver and if you request a mode that isn't currently possible, it will return -1. The available modes are:

- SWITCH_NONE
  Disables switching. This is the default in single-tasking systems like DOS. It may be supported on other platforms, but you should use it with caution, because your users won't be impressed if they want to tab away from your program, but you don't let them!
- SWITCH_PAUSE
  Pauses the program whenever it is in the background. Execution will be resumed as soon as the user switches back to it. This is the default in most fullscreen multitasking environments, for example the Linux console, but not under Windows.

- SWITCH_AMNESIA
  Like SWITCH_PAUSE, but this mode doesn't bother to remember the contents of video memory, so the screen, and any video bitmaps that you have created, will be erased after the user switches away and then back to your program. This is not a terribly useful mode to have, but it is the default for the fullscreen drivers under Windows because DirectDraw is too dumb to implement anything better.
- SWITCH_BACKGROUND
  The program will carry on running in the background, with the screen bitmap temporarily being pointed at a memory buffer for the fullscreen drivers. You must take special care when using this mode, because bad things will happen if the screen bitmap gets changed around when your program isn't expecting it (see below).
- SWITCH_BACKAMNESIA
  Like SWITCH_BACKGROUND, but this mode doesn't bother to remember the contents of video memory (see SWITCH_AMNESIA). It is again the only mode supported by the fullscreen drivers under Windows that lets the program keep running in the background.

Note that you should be very careful when you are using graphics routines in the switching context: you must always call acquire_screen() before the start of any drawing code onto the screen and not release it until you are completely finished, because the automatic locking mechanism may not be good enough to work when the program runs in the background or has just been raised in the foreground.

See also:

See Section 8.8 [set_display_switch_callback], page 73.

See Section 8.10 [get_display_switch_mode], page 74.

8.8 set_display_switch_callback

`int set_display_switch_callback(int dir, void (*cb)());`

Installs a notification callback for the switching mode that was previously selected by calling set_display_switch_mode(). The direction parameter can either be SWITCH_IN or SWITCH_OUT, depending whether you want to be notified about switches away from your program or back to your program. You can sometimes install callbacks for both directions at the same time, but not every platform supports this, so this function may return -1 if your request is impossible. You can install several switch callbacks at the same time.

See also:

See Section 8.9 [remove_display_switch_callback], page 74.

See Section 8.7 [set_display_switch_mode], page 72.

8.9 remove_display_switch_callback

`void remove_display_switch_callback(void (*cb)());`

Removes a notification callback that was previously installed by calling set_display_switch_callback(). All the callbacks will automatically be removed when you call set_display_switch_mode().

See also:

See Section 8.8 [set_display_switch_callback], page 73.

8.10 get_display_switch_mode

`int get_display_switch_mode();`

Returns the current display switching mode, in the same format passed to set_display_switch_mode().

See also:

See Section 8.7 [set_display_switch_mode], page 72.

8.11 gfx_capabilities

`extern int gfx_capabilities;`

Bitfield describing the capabilities of the current graphics driver and video hardware. This may contain combination any of the flags:

GFX_CAN_SCROLL:
Indicates that the scroll_screen() function may be used with this driver.

GFX_CAN_TRIPLE_BUFFER:
Indicates that the request_scroll() and poll_scroll() functions may be used with this driver. If this flag is not set, it is possible that the enable_triple_buffer() function may be able to activate it.

GFX_HW_CURSOR:
Indicates that a hardware mouse cursor is in use. When this flag is set, it is safe to draw onto the screen without hiding the mouse pointer first. Note that not every cursor graphic can be implemented in hardware: in particular VBE/AF only supports 2-color images up to 32x32 in size, where the second color is an exact inverse of the first. This means that Allegro may need to switch between hardware and software cursors at any point during the execution of your program, so you should not assume that this flag will remain constant for long periods of time. It only tells you whether a hardware cursor is in use at the current time, and may change whenever you hide/redisplay the pointer.

GFX_HW_HLINE:
Indicates that the normal opaque version of the hline() function is implemented using a hardware accelerator. This will improve the performance not only of hline() itself, but also of many other functions that use it as a workhorse, for example circlefill(), triangle(), and floodfill().

GFX_HW_HLINE_XOR:
Indicates that the XOR version of the hline() function, and any other functions that use it as a workhorse, are implemented using a hardware accelerator.

GFX_HW_HLINE_SOLID_PATTERN:
Indicates that the solid and masked pattern modes of the hline() function, and any other functions that use it as a workhorse, are implemented using a hardware accelerator (see note below).

GFX_HW_HLINE_COPY_PATTERN:
Indicates that the copy pattern mode of the hline() function, and any other functions that use it as a workhorse, are implemented using a hardware accelerator (see note below).

GFX_HW_FILL:
Indicates that the opaque version of the rectfill() function, the clear_bitmap() routine, and clear_to_color(), are implemented using a hardware accelerator.

GFX_HW_FILL_XOR:
Indicates that the XOR version of the rectfill() function is implemented using a hardware accelerator.

GFX_HW_FILL_SOLID_PATTERN:
Indicates that the solid and masked pattern modes of the rectfill() function are implemented using a hardware accelerator (see note below).

GFX_HW_FILL_COPY_PATTERN:
Indicates that the copy pattern mode of the rectfill() function is implemented using a hardware accelerator (see note below).

GFX_HW_LINE:
Indicates that the opaque mode line() and vline() functions are implemented using a hardware accelerator.

GFX_HW_LINE_XOR:
Indicates that the XOR version of the line() and vline() functions are implemented using a hardware accelerator.

GFX_HW_TRIANGLE:
Indicates that the opaque mode triangle() function is implemented using a hardware accelerator.

GFX_HW_TRIANGLE_XOR:
Indicates that the XOR version of the triangle() function is implemented using a hardware accelerator.

GFX_HW_GLYPH:
Indicates that monochrome character expansion (for text drawing) is implemented using a hardware accelerator.

GFX_HW_VRAM_BLIT:
Indicates that blitting from one part of the screen to another is implemented using a hardware accelerator. If this flag is set, blitting within the video memory will almost certainly be the fastest possible way to display an image, so it may be worth storing some of your more frequently used graphics in an offscreen portion of the video memory.

GFX_HW_VRAM_BLIT_MASKED:
Indicates that the masked_blit() routine is capable of a hardware accelerated copy from one part of video memory to another, and that draw_sprite() will use a hardware copy when given a sub-bitmap of the screen or a video memory bitmap as the source image. If this flag is set, copying within the video memory will almost certainly be the fastest possible way to display an image, so it may be worth storing some of your more frequently used sprites in an offscreen portion of the video memory.

Warning: if this flag is not set, masked_blit() and draw_sprite() will not work correctly when used with a video memory source image! You must only try to use these functions to copy within the video memory if they are supported in hardware.

GFX_HW_MEM_BLIT:
Indicates that blitting from a memory bitmap onto the screen is being accelerated in hardware.

GFX_HW_MEM_BLIT_MASKED:
Indicates that the masked_blit() and draw_sprite() functions are being accelerated in hardware when the source image is a memory bitmap and the destination is the physical screen.

GFX_HW_SYS_TO_VRAM_BLIT:
Indicates that blitting from a system bitmap onto the screen is being accelerated in hardware. Note that some acceleration may be present even if this flag is not set, because system bitmaps can benefit from normal memory to screen blitting as well. This flag will only be set if system bitmaps have further acceleration above and beyond what is provided by GFX_HW_MEM_BLIT.

GFX_HW_SYS_TO_VRAM_BLIT_MASKED:
Indicates that the masked_blit() and draw_sprite() functions are being accelerated in hardware when the source image is a system bitmap and the destination is the physical screen. Note that some acceleration may be present even if this flag is not set, because system bitmaps can benefit from normal memory to screen blitting as well. This flag will only be set if system bitmaps have further acceleration above and beyond what is provided by GFX_HW_MEM_BLIT_MASKED.

Note: even if the capabilities information says that patterned drawing is supported by the hardware, it will not be possible for every size of pattern. VBE/AF only supports patterns up to 8x8 in size, so Allegro will fall back on the original non-accelerated drawing routines whenever you use a pattern larger than this.

Note2: these hardware acceleration features will only take effect when you are drawing directly onto the screen bitmap, a video memory bitmap, or a sub-bitmap thereof. Accelerated hardware is most useful in a page flipping or triple buffering setup, and is unlikely to make any difference to the classic "draw onto a memory bitmap, then blit to the screen" system.

See also:

See Section 9.1 [screen], page 80.

See Section 9.5 [create_video_bitmap], page 82.
See Section 8.13 [scroll_screen], page 77.
See Section 8.14 [request_scroll], page 78.
See Section 4.7 [show_mouse], page 44.
See Section 8.12 [enable_triple_buffer], page 77.

8.12 enable_triple_buffer

`int enable_triple_buffer();`

If the GFX_CAN_TRIPLE_BUFFER bit of the gfx_capabilities field is not set, you can attempt to enable it by calling this function. In particular if you are running in mode-X in a clean DOS environment, this routine will enable the timer retrace simulator, which will activate the triple buffering functions. Returns zero if triple buffering is enabled.

See also:
See Section 8.11 [gfx_capabilities], page 74.
See Section 5.10 [timer_simulate_retrace], page 52.
See Section 8.14 [request_scroll], page 78.
See Section 8.17 [request_video_bitmap], page 79.

8.13 scroll_screen

`int scroll_screen(int x, int y);`

Attempts to scroll the hardware screen to display a different part of the virtual screen (initially it will be positioned at 0, 0, which is the top left corner). Returns zero on success: it may fail if the graphics driver can't handle hardware scrolling or the virtual screen isn't large enough. You can use this to move the screen display around in a large virtual screen space, or to page flip back and forth between two non-overlapping areas of the virtual screen. Note that to draw outside the original position in the screen bitmap you will have to alter the clipping rectangle: see below.

Mode-X scrolling is reliable and will work on any card. Unfortunately most VESA implementations can only handle horizontal scrolling in four pixel increments, so smooth horizontal panning is impossible in SVGA modes. This is a shame, but I can't see any way round it. A significant number of VESA implementations seem to be very buggy when it comes to scrolling in truecolor video modes, so I suggest you don't depend on this routine working correctly in the truecolor resolutions unless you can be sure that SciTech Display Doctor is installed.

Allegro will handle any necessary vertical retrace synchronisation when scrolling the screen, so you don't need to call vsync() before it. This means that scroll_screen() has the same time delay effects as vsync().

See also:

See Section 8.6 [set_gfx_mode], page 70.

See Section 8.16 [show_video_bitmap], page 78.

See Section 8.14 [request_scroll], page 78.

See Section 8.17 [request_video_bitmap], page 79.

8.14 request_scroll

`int request_scroll(int x, int y);`

This function is used for triple buffering. It requests a hardware scroll to the specified position, but returns immediately rather than waiting for a retrace. The scroll will then take place during the next vertical retrace, but you can carry on running other code in the meantime and use the poll_scroll() routine to detect when the flip has actually taken place (see examples/ex3buf.c). Triple buffering is only possible on certain hardware: it will work in any mode-X resolution if the timer retrace simulator is active (but this doesn't work correctly under win95), plus it is supported by the VBE 3.0 and VBE/AF drivers for a limited number of high-end graphics cards. You can look at the GFX_CAN_TRIPLE_BUFFER bit in the gfx_capabilities flag to see if it will work with the current driver. This function returns zero on success.

See also:

See Section 8.15 [poll_scroll], page 78.

See Section 8.17 [request_video_bitmap], page 79.

See Section 8.11 [gfx_capabilities], page 74.

See Section 5.10 [timer_simulate_retrace], page 52.

See Section 8.13 [scroll_screen], page 77.

8.15 poll_scroll

`int poll_scroll();`

This function is used for triple buffering. It checks the status of a hardware scroll previously initiated by the request_scroll() routine, returning non-zero if it is still waiting to take place, and zero if it has already happened.

See also:

See Section 8.14 [request_scroll], page 78.

See Section 8.17 [request_video_bitmap], page 79.

8.16 show_video_bitmap

`int show_video_bitmap(BITMAP *bitmap);`

Attempts to page flip the hardware screen to display the specified video bitmap object, which must be the same size as the physical screen, and should have been obtained by calling the create_video_bitmap() function. Returns zero on success and non-zero on failure.

Allegro will handle any necessary vertical retrace synchronisation when page flipping, so you don't need to call vsync() before it. This means that show_video_bitmap() has the same time delay effects as vsync().

See also:

See Section 8.13 [scroll_screen], page 77.

See Section 9.5 [create_video_bitmap], page 82.

8.17 request_video_bitmap

`int request_video_bitmap(BITMAP *bitmap);`

This function is used for triple buffering. It requests a page flip to display the specified video bitmap object, but returns immediately rather than waiting for a retrace. The flip will then take place during the next vertical retrace, but you can carry on running other code in the meantime and use the poll_scroll() routine to detect when the flip has actually taken place. Triple buffering is only possible on certain hardware: see the comments about request_scroll(). Returns zero on success.

See also:
See Section 8.15 [poll_scroll], page 78.
See Section 8.14 [request_scroll], page 78.
See Section 8.11 [gfx_capabilities], page 74.
See Section 5.10 [timer_simulate_retrace], page 52.
See Section 9.5 [create_video_bitmap], page 82.
See Section 8.13 [scroll_screen], page 77.

9 Bitmap objects

Once you have selected a graphics mode, you can draw things onto the display via the 'screen' bitmap. All the Allegro graphics routines draw onto BITMAP structures, which are areas of memory containing rectangular images, stored as packed byte arrays (in 8 bit modes one byte per pixel, in 15 and 16 bit modes sizeof(short) bytes per pixel, in 24 bit modes 3 bytes per pixel and in 32 bit modes sizeof(long) bytes per pixel). You can create and manipulate bitmaps in system RAM, or you can write to the special 'screen' bitmap which represents the video memory in your graphics card.

For example, to draw a pixel onto the screen you would write:

```
     putpixel(screen, x, y, color);
```

Or to implement a double-buffered system:

```
     BITMAP *bmp = create_bitmap(320, 200);    // make a bitmap in system RAM
     clear_bitmap(bmp);                        // zero the memory bitmap
     putpixel(bmp, x, y, color);               // draw onto the memory bitmap
     blit(bmp, screen, 0, 0, 0, 0, 320, 200);  // copy it to the screen
```

See below for information on how to get direct access to the image memory in a bitmap.

Allegro supports several different types of bitmaps:

- The screen bitmap, which represents the hardware video memory. Ultimately you have to draw onto this in order for your image to be visible. It is destroyed by any subsequent calls to set_gfx_mode().

- Memory bitmaps, which are located in system RAM and can be used to store graphics or as temporary drawing spaces for double buffered systems. These can be obtained by calling create_bitmap(), load_pcx(), or by loading a grabber datafile.

- Sub-bitmaps. These share image memory with a parent bitmap (which can be the screen, a memory bitmap, or another sub-bitmap), so drawing onto them will also change their parent. They can be of any size and located anywhere within the parent bitmap, and can have their own clipping rectangles, so they are a useful way of dividing a bitmap into several smaller units, eg. splitting a large virtual screen into multiple sections (see examples/exscroll.c).

- Video memory bitmaps. These are created by the create_video_bitmap() function, and are usually implemented as sub-bitmaps of the screen object. They must be destroyed by destroy_bitmap() before any subsequent calls to set_gfx_mode().

- System bitmaps. These are created by the create_system_bitmap() function, and are a sort of halfway house between memory and video bitmaps. They live in system memory, so you aren't limited by the amount of video ram in your card, but they are stored in a platform-specific format that may enable better hardware acceleration than is possible with a normal memory bitmap (see the GFX_HW_SYS_TO_VRAM_BLIT and GFX_HW_SYS_TO_VRAM_BLIT_MASKED flags in gfx_capabilities). System bitmaps must be accessed in the same way as video bitmaps, using the bank switch functions and bmp_write*() macros. Not every platform implements this type of bitmap: if they aren't available, create_system_bitmap() will function identically to create_bitmap(). They must be destroyed by destroy_bitmap() before any subsequent calls to set_gfx_mode().

9.1 screen

`extern BITMAP *screen;`

Global pointer to a bitmap, sized VIRTUAL_W x VIRTUAL_H. This is created by set_gfx_mode(), and represents the hardware video memory. Only a part of this bitmap will actually be visible, sized SCREEN_W x SCREEN_H. Normally this is the top left corner of the larger virtual screen, so you can ignore the extra invisible virtual size of the bitmap if you aren't interested in hardware scrolling or page flipping. To move the visible window to other parts of the screen bitmap, call scroll_screen(). Initially the clipping rectangle will be limited to the physical screen size, so if you want to draw onto a larger virtual screen space outside this rectangle, you will need to adjust the clipping.

See also:
See Section 8.6 [set_gfx_mode], page 70.
See Section 9.15 [is_screen_bitmap], page 85.
See Section 9.5 [create_video_bitmap], page 82.
See Section 8.13 [scroll_screen], page 77.

9.2 create_bitmap

`BITMAP *create_bitmap(int width, int height);`

Creates a memory bitmap sized width by height, and returns a pointer to it. The bitmap will have clipping turned on, and the clipping rectangle set to the full size of the bitmap. The image memory will not be cleared, so it will probably contain garbage: you should clear the bitmap before using it. This routine always uses the global pixel format, as specified by calling set_color_depth().

See also:
See Section 9.3 [create_bitmap_ex], page 81.
See Section 9.4 [create_sub_bitmap], page 81.
See Section 9.5 [create_video_bitmap], page 82.
See Section 9.6 [create_system_bitmap], page 82.
See Section 9.7 [destroy_bitmap], page 83.
See Section 8.1 [set_color_depth], page 68.
See Section 9.14 [is_memory_bitmap], page 84.

9.3 create_bitmap_ex

`BITMAP *create_bitmap_ex(int color_depth, int width, int height);`

Creates a bitmap in a specific color depth (8, 15, 16, 24 or 32 bits per pixel).

See also:
See Section 9.2 [create_bitmap], page 81.
See Section 9.4 [create_sub_bitmap], page 81.
See Section 9.5 [create_video_bitmap], page 82.
See Section 9.6 [create_system_bitmap], page 82.
See Section 9.7 [destroy_bitmap], page 83.
See Section 9.14 [is_memory_bitmap], page 84.

9.4 create_sub_bitmap

```
BITMAP *create_sub_bitmap(BITMAP *parent, int x, y, width, height);
```

Creates a sub-bitmap, ie. a bitmap sharing drawing memory with a pre-existing bitmap, but possibly with a different size and clipping settings. When creating a sub-bitmap of the mode-X screen, the x position must be a multiple of four. The sub-bitmap width and height can extend beyond the right and bottom edges of the parent (they will be clipped), but the origin point must lie within the parent region.

See also:

See Section 9.2 [create_bitmap], page 81.

See Section 9.3 [create_bitmap_ex], page 81.

See Section 9.7 [destroy_bitmap], page 83.

See Section 9.18 [is_sub_bitmap], page 85.

9.5 create_video_bitmap

```
BITMAP *create_video_bitmap(int width, int height);
```

Allocates a video memory bitmap of the specified size, returning a pointer to it on success or NULL on failure (ie. if you have run out of vram). This can be used to allocate offscreen video memory for storing source graphics ready for a hardware accelerated blitting operation, or to create multiple video memory pages which can then be displayed by calling show_video_bitmap(). Video memory bitmaps are usually allocated from the same space as the screen bitmap, so they may overlap with it: it is not therefore a good idea to use the global screen at the same time as any surfaces returned by this function.

See also:

See Section 9.2 [create_bitmap], page 81.

See Section 9.3 [create_bitmap_ex], page 81.

See Section 9.6 [create_system_bitmap], page 82.

See Section 9.4 [create_sub_bitmap], page 81.

See Section 9.7 [destroy_bitmap], page 83.

See Section 9.1 [screen], page 80.

See Section 8.16 [show_video_bitmap], page 78.

See Section 8.11 [gfx_capabilities], page 74.

See Section 9.16 [is_video_bitmap], page 85.

9.6 create_system_bitmap

```
BITMAP *create_system_bitmap(int width, int height);
```

Allocates a system memory bitmap of the specified size, returning a pointer to it on success or NULL on failure.

See also:
See Section 9.2 [create_bitmap], page 81.
See Section 9.3 [create_bitmap_ex], page 81.
See Section 9.5 [create_video_bitmap], page 82.
See Section 9.4 [create_sub_bitmap], page 81.
See Section 9.7 [destroy_bitmap], page 83.
See Section 9.17 [is_system_bitmap], page 85.

9.7 destroy_bitmap

`void destroy_bitmap(BITMAP *bitmap);`

Destroys a memory bitmap, sub-bitmap, video memory bitmap, or system bitmap when you are finished with it.

See also:
See Section 9.2 [create_bitmap], page 81.
See Section 10.1 [load_bitmap], page 87.

9.8 lock_bitmap

`void lock_bitmap(BITMAP *bitmap);`

Under DOS, locks all the memory used by a bitmap. You don't normally need to call this function unless you are doing very weird things in your program.

9.9 bitmap_color_depth

`int bitmap_color_depth(BITMAP *bmp);`

Returns the color depth of the specified bitmap (8, 15, 16, 24, or 32). Note that calling it on the 'screen' bitmap will return the pixel format currently in use, as specified by the latest call to set_color_depth(), once a graphics mode has been set.

See also:
See Section 8.1 [set_color_depth], page 68.
See Section 9.10 [bitmap_mask_color], page 83.

9.10 bitmap_mask_color

`int bitmap_mask_color(BITMAP *bmp);`

Returns the mask color for the specified bitmap (the value which is skipped when drawing sprites). For 256 color bitmaps this is zero, and for truecolor bitmaps it is bright pink (maximum red and blue, zero green).

See also:
See Section 12.12 [MASK_COLOR_8], page 104.
See Section 8.1 [set_color_depth], page 68.
See Section 9.9 [bitmap_color_depth], page 83.

9.11 is_same_bitmap

`int is_same_bitmap(BITMAP *bmp1, BITMAP *bmp2);`

Returns TRUE if the two bitmaps describe the same drawing surface, ie. the pointers are equal, one is a sub-bitmap of the other, or they are both sub-bitmaps of a common parent.

See also:
See Section 9.4 [create_sub_bitmap], page 81.

9.12 is_linear_bitmap

`int is_linear_bitmap(BITMAP *bmp);`

Returns TRUE if bmp is a linear bitmap, ie. a memory bitmap, mode 13h screen, or SVGA screen. Linear bitmaps can be used with the _putpixel(), _getpixel(), bmp_write_line(), and bmp_read_line() functions.

See also:
See Section 9.13 [is_planar_bitmap], page 84.
See Section 9.14 [is_memory_bitmap], page 84.

9.13 is_planar_bitmap

`int is_planar_bitmap(BITMAP *bmp);`

Returns TRUE if bmp is a planar (mode-X or Xtended mode) screen bitmap.

See also:
See Section 9.12 [is_linear_bitmap], page 84.
See Section 9.14 [is_memory_bitmap], page 84.

9.14 is_memory_bitmap

`int is_memory_bitmap(BITMAP *bmp);`

Returns TRUE if bmp is a memory bitmap, ie. it was created by calling create_bitmap() or loaded from a grabber datafile or image file. Memory bitmaps can be accessed directly via the line pointers in the bitmap structure, eg. bmp->line[y][x] = color.

See also:

See Section 9.12 [is_linear_bitmap], page 84.

See Section 9.13 [is_planar_bitmap], page 84.

9.15 is_screen_bitmap

`int is_screen_bitmap(BITMAP *bmp);`

Returns TRUE if bmp is the screen bitmap, or a sub-bitmap of the screen.

See also:

See Section 9.1 [screen], page 80.

See Section 9.4 [create_sub_bitmap], page 81.

9.16 is_video_bitmap

`int is_video_bitmap(BITMAP *bmp);`

Returns TRUE if bmp is the screen bitmap, a video memory bitmap, or a sub-bitmap of either.

See also:

See Section 9.1 [screen], page 80.

See Section 9.5 [create_video_bitmap], page 82.

See Section 9.4 [create_sub_bitmap], page 81.

9.17 is_system_bitmap

`int is_system_bitmap(BITMAP *bmp);`

Returns TRUE if bmp is a system bitmap object, or a sub-bitmap of one.

See also:

See Section 9.6 [create_system_bitmap], page 82.

See Section 9.4 [create_sub_bitmap], page 81.

9.18 is_sub_bitmap

`int is_sub_bitmap(BITMAP *bmp);`

Returns TRUE if bmp is a sub-bitmap.

See also:

See Section 9.4 [create_sub_bitmap], page 81.

9.19 acquire_bitmap

`void acquire_bitmap(BITMAP *bmp);`

Locks the specified video memory bitmap prior to drawing onto it. This does not apply to memory bitmaps, and only affects some platforms (Windows needs it, DOS does not). These calls are not strictly required, because the drawing routines will automatically acquire the bitmap before accessing it, but locking a DirectDraw surface is very slow, so you will get much better performance if you acquire the screen just once at the start of your main redraw function, and only release it when the drawing is completely finished. Multiple acquire calls may be nested, and the bitmap will only be truly released when the lock count returns to zero. Be warned that DirectX programs activate a mutex lock whenever a surface is locked, which prevents them from getting any input messages, so you must be sure to release all your bitmaps before using any timer, keyboard, or other non-graphics routines!

See also:

See Section 9.20 [release_bitmap], page 86.

See Section 9.21 [acquire_screen], page 86.

See Section 9.22 [release_screen], page 87.

9.20 release_bitmap

`void release_bitmap(BITMAP *bmp);`

Releases a bitmap that was previously locked by calling acquire_bitmap(). If the bitmap was locked multiple times, you must release it the same number of times before it will truly be unlocked.

See also:

See Section 9.19 [acquire_bitmap], page 86.

See Section 9.21 [acquire_screen], page 86.

See Section 9.22 [release_screen], page 87.

9.21 acquire_screen

`void acquire_screen();`

Shortcut version of acquire_bitmap(screen);

See also:

See Section 9.19 [acquire_bitmap], page 86.

See Section 9.20 [release_bitmap], page 86.

See Section 9.22 [release_screen], page 87.

9.22 release_screen

`void release_screen();`

Shortcut version of release_bitmap(screen);

See also:

See Section 9.19 [acquire_bitmap], page 86.

See Section 9.20 [release_bitmap], page 86.

See Section 9.21 [acquire_screen], page 86.

9.23 set_clip

`void set_clip(BITMAP *bitmap, int x1, int y1, int x2, int y2);`

Each bitmap has an associated clipping rectangle, which is the area of the image that it is ok to draw on. Nothing will be drawn to positions outside this space. Pass the two opposite corners of the clipping rectangle: these are inclusive, eg. set_clip(bitmap, 16, 16, 32, 32) will allow drawing to (16, 16) and (32, 32), but not to (15, 15) and (33, 33). If x1, y1, x2, and y2 are all zero, clipping will be turned off, which may slightly speed up some drawing operations (usually a negligible difference, although every little helps) but will result in your program dying a horrible death if you try to draw beyond the edges of the bitmap.

10 Loading image files

Warning: when using truecolor images, you should always set the graphics mode before loading any bitmap data! Otherwise the pixel format (RGB or BGR) will not be known, so the file may be converted wrongly.

10.1 load_bitmap

`BITMAP *load_bitmap(const char *filename, RGB *pal);`

Loads a bitmap from a file, returning a pointer to a bitmap and storing the palette data in the specified location, which should be an array of 256 RGB structures. You are responsible for destroying the bitmap when you are finished with it. Returns NULL on error. At present this function supports BMP, LBM, PCX, and TGA files, determining the type from the file extension.

If the file contains a truecolor image, you must set the video mode or call set_color_conversion() before loading it. In this case, if the destination color depth is 8-bit, the palette will be generated by calling generate_optimized_palette() on the bitmap; otherwise, the returned palette will be generated by calling generate_332_palette().

The pal argument may be NULL. In this case, the palette data are simply not returned. Additionally, if the file is a truecolor image and the destination color depth is 8-bit, the color conversion process will use the current palette instead of generating an optimized one.

See also:
See Section 10.2 [load_bmp], page 88.
See Section 10.3 [load_lbm], page 88.
See Section 10.4 [load_pcx], page 88.
See Section 10.5 [load_tga], page 88.
See Section 9.7 [destroy_bitmap], page 83.
See Section 10.6 [save_bitmap], page 89.
See Section 10.10 [register_bitmap_file_type], page 90.
See Section 8.1 [set_color_depth], page 68.
See Section 10.11 [set_color_conversion], page 90.
See Section 11.19 [generate_optimized_palette], page 98.
See Section 11.18 [generate_332_palette], page 97.

10.2 load_bmp

```
BITMAP *load_bmp(const char *filename, RGB *pal);
```

Loads a 256 color or 24 bit truecolor Windows or OS/2 BMP file.

See also:
See Section 10.1 [load_bitmap], page 87.

10.3 load_lbm

```
BITMAP *load_lbm(const char *filename, RGB *pal);
```

Loads a 256 color IFF ILBM/PBM file.

See also:
See Section 10.1 [load_bitmap], page 87.

10.4 load_pcx

```
BITMAP *load_pcx(const char *filename, RGB *pal);
```

Loads a 256 color or 24 bit truecolor PCX file.

See also:
See Section 10.1 [load_bitmap], page 87.

10.5 load_tga

```
BITMAP *load_tga(const char *filename, RGB *pal);
```

Loads a 256 color, 15 bit hicolor, 24 bit truecolor, or 32 bit truecolor+alpha TGA file.

See also:

See Section 10.1 [load_bitmap], page 87.

10.6 save_bitmap

`int save_bitmap(const char *filename, BITMAP *bmp, const RGB *pal);`

Writes a bitmap into a file, using the specified palette, which should be an array of 256 RGB structures. Returns non-zero on error. The output format is determined from the filename extension: at present this function supports BMP, PCX and TGA formats.

Two things to watch out for: on some video cards it may be faster to copy the screen to a memory bitmap and save the latter, and if you use this to dump the screen into a file you may end up with an image much larger than you were expecting, because Allegro often creates virtual screens larger than the visible screen. You can get around this by using a sub-bitmap to specify which part of the screen to save, eg:

```
BITMAP *bmp;
PALETTE pal;

get_palette(pal);
bmp = create_sub_bitmap(screen, 0, 0, SCREEN_W, SCREEN_H);
save_bitmap("dump.pcx", bmp, pal);
destroy_bitmap(bmp);
```

See also:

See Section 10.7 [save_bmp], page 89.

See Section 10.8 [save_pcx], page 89.

See Section 10.9 [save_tga], page 90.

See Section 10.1 [load_bitmap], page 87.

See Section 10.10 [register_bitmap_file_type], page 90.

10.7 save_bmp

`int save_bmp(const char *filename, BITMAP *bmp, const RGB *pal);`

Writes a bitmap into a 256 color or 24 bit truecolor BMP file.

See also:

See Section 10.6 [save_bitmap], page 89.

10.8 save_pcx

`int save_pcx(const char *filename, BITMAP *bmp, const RGB *pal);`

Writes a bitmap into a 256 color or 24 bit truecolor PCX file.

See also:

See Section 10.6 [save_bitmap], page 89.

10.9 save_tga

```
int save_tga (const char *filename, BITMAP *bmp, const RGB *pal);
```

Writes a bitmap into a 256 color, 15 bit hicolor, 24 bit truecolor, or 32 bit truecolor+alpha TGA file.

See also:

See Section 10.6 [save_bitmap], page 89.

10.10 register_bitmap_file_type

```
void register_bitmap_file_type(const char *ext, BITMAP *(*load)(const char
*filename, RGB *pal), int (*save)(const char *filename, BITMAP *bmp, const
RGB *pal));
```

Informs the load_bitmap() and save_bitmap() functions of a new file type, providing routines to read and write images in this format (either function may be NULL).

See also:

See Section 10.1 [load_bitmap], page 87.

See Section 10.6 [save_bitmap], page 89.

10.11 set_color_conversion

```
void set_color_conversion(int mode);
```

Specifies how to convert images between the various color depths when reading graphics from external bitmap files or datafiles. The mode is a bitmask specifying which types of conversion are allowed. If the appropriate bit is set, data will be converted into the current pixel format (selected by calling the set_color_depth() function), otherwise it will be left in the same format as the disk file, leaving you to convert it manually before the graphic can be displayed. The default mode is total conversion, so that all images will be loaded in the appropriate format for the current video mode. Valid bit flags are:

```
COLORCONV_NONE                  // disable all format
                                // conversions
COLORCONV_8_TO_15               // expand 8 bits to 15 bits
COLORCONV_8_TO_16               // expand 8 bits to 16 bits
COLORCONV_8_TO_24               // expand 8 bits to 24 bits
COLORCONV_8_TO_32               // expand 8 bits to 32 bits
COLORCONV_15_TO_8               // reduce 15 bits to 8 bits
```

```
     COLORCONV_15_TO_16           // expand 15 bits to 16 bits
     COLORCONV_15_TO_24           // expand 15 bits to 24 bits
     COLORCONV_15_TO_32           // expand 15 bits to 32 bits
     COLORCONV_16_TO_8            // reduce 16 bits to 8 bits
     COLORCONV_16_TO_15           // reduce 16 bits to 15 bits
     COLORCONV_16_TO_24           // expand 16 bits to 24 bits
     COLORCONV_16_TO_32           // expand 16 bits to 32 bits
     COLORCONV_24_TO_8            // reduce 24 bits to 8 bits
     COLORCONV_24_TO_15           // reduce 24 bits to 15 bits
     COLORCONV_24_TO_16           // reduce 24 bits to 16 bits
     COLORCONV_24_TO_32           // expand 24 bits to 32 bits
     COLORCONV_32_TO_8            // reduce 32 bit RGB to 8 bits
     COLORCONV_32_TO_15           // reduce 32 bit RGB to 15 bits
     COLORCONV_32_TO_16           // reduce 32 bit RGB to 16 bits
     COLORCONV_32_TO_24           // reduce 32 bit RGB to 24 bits
     COLORCONV_32A_TO_8           // reduce 32 bit RGBA to 8 bits
     COLORCONV_32A_TO_15          // reduce 32 bit RGBA to 15 bits
     COLORCONV_32A_TO_16          // reduce 32 bit RGBA to 16 bits
     COLORCONV_32A_TO_24          // reduce 32 bit RGBA to 24 bits
     COLORCONV_DITHER_PAL         // dither when reducing to 8 bit
     COLORCONV_DITHER_HI          // dither when reducing to
                                  // hicolor
     COLORCONV_KEEP_TRANS         // keep original transparency
```

For convenience, the following macros can be used to select common combinations of these flags:

```
     COLORCONV_EXPAND_256         // expand 256 colors to
                                  // hi/truecolor
     COLORCONV_REDUCE_TO_256      // reduce hi/truecolor to 256
                                  // colors
     COLORCONV_EXPAND_15_TO_16    // expand 15 bit hicolor to
                                  // 16 bits
     COLORCONV_REDUCE_16_TO_15    // reduce 16 bit hicolor to
                                  // 15 bits
     COLORCONV_EXPAND_HI_TO_TRUE  // expand 15/16 bits to
                                  // 24/32 bits
     COLORCONV_REDUCE_TRUE_TO_HI  // reduce 24/32 bits to
                                  // 15/16 bits
     COLORCONV_24_EQUALS_32       // convert between 24 and
                                  // 32 bits
     COLORCONV_TOTAL              // everything to current format
     COLORCONV_PARTIAL            // convert 15 <-> 16 and
                                  // 24 <-> 32
     COLORCONV_MOST               // all but hi/truecolor <-> 256
     COLORCONV_DITHER             // dither during all color
                                  // reductions
```

If you enable the COLORCONV_DITHER flag, dithering will be performed whenever truecolor graphics are converted into a hicolor or paletted format, including by the blit() function, and any automatic conversions that take place while reading graphics from disk. This can produce much better looking results, but is obviously slower than a direct conversion.

If you intend using converted bitmaps with functions like masked_blit() or draw_sprite(), you should specify the COLORCONV_KEEP_TRANS flag. It will ensure that the masked areas in the bitmap before and after the conversion stay exactly the same, by mapping transparent colors to each other and adjusting colors which would be converted to the transparent color otherwise. It affects every blit() operation between distinct pixel formats and every automatic conversion.

See also:
See Section 8.1 [set_color_depth], page 68.
See Section 10.1 [load_bitmap], page 87.
See Section 29.1 [load_datafile], page 191.
See Section 29.9 [fixup_datafile], page 193.
See Section 12.6 [makecol15_dither], page 101.

11 Palette routines

All the Allegro drawing functions use integer parameters to represent colors. In truecolor resolutions these numbers encode the color directly as a collection of red, green, and blue bits, but in a regular 256 color mode the values are treated as indexes into the current palette, which is a table listing the red, green and blue intensities for each of the 256 possible colors.

Palette entries are stored in an RGB structure, which contains red, green and blue intensities in the VGA hardware format, ranging from 0-63, and is defined as:

```
typedef struct RGB
{
   unsigned char r, g, b;
} RGB;
```

For example:

```
RGB black = { 0,  0,  0  };
RGB white = { 63, 63, 63 };
RGB green = { 0,  63, 0  };
RGB grey  = { 32, 32, 32 };
```

The type PALETTE is defined to be an array of 256 RGB structures.

You may notice that a lot of the code in Allegro spells 'palette' as 'pallete'. This is because the headers from my old Mark Williams compiler on the Atari spelt it with two l's, so that is what I'm used to. Allegro will happily accept either spelling, due to some #defines in allegro/alcompat.h.

11.1 vsync

`void vsync();`

Waits for a vertical retrace to begin. The retrace happens when the electron beam in your monitor has reached the bottom of the screen and is moving back to the top ready for another scan. During this short period the graphics card isn't sending any data to the monitor, so you can do things to it that aren't possible at other times, such as altering the palette without causing flickering (snow). Allegro will automatically wait for a retrace before altering the palette or doing any hardware scrolling, though, so you don't normally need to bother with this function.

See also:

See Section 11.4 [set_palette], page 94.

See Section 8.13 [scroll_screen], page 77.

See Section 5.10 [timer_simulate_retrace], page 52.

11.2 set_color

`void set_color(int index, const RGB *p);`

Sets the specified palette entry to the specified RGB triplet. Unlike the other palette functions this doesn't do any retrace synchronisation, so you should call vsync() before it to prevent snow problems.

See also:

See Section 11.4 [set_palette], page 94.

See Section 11.6 [get_color], page 94.

See Section 11.3 [_set_color], page 93.

11.3 _set_color

`void _set_color(int index, const RGB *p);`

This is an inline version of set_color(), intended for use in the vertical retrace simulator callback function. It should only be used in VGA mode 13h and mode-X, because some of the more recent SVGA chipsets aren't VGA compatible (set_color() and set_palette() will use VESA calls on these cards, but _set_color() doesn't know about that).

See also:

See Section 11.2 [set_color], page 93.

See Section 8.6 [set_gfx_mode], page 70.

See Section 5.10 [timer_simulate_retrace], page 52.

11.4 set_palette

`void set_palette(const PALETTE p);`

Sets the entire palette of 256 colors. You should provide an array of 256 RGB structures. Unlike set_color(), there is no need to call vsync() before this function.

See also:

See Section 8.6 [set_gfx_mode], page 70.

See Section 11.5 [set_palette_range], page 94.

See Section 11.2 [set_color], page 93.

See Section 11.7 [get_palette], page 94.

See Section 11.16 [select_palette], page 97.

See Section 12.11 [palette_color], page 103.

11.5 set_palette_range

`void set_palette_range(const PALETTE p, int from, int to, int vsync);`

Sets the palette entries between from and to (inclusive: pass 0 and 255 to set the entire palette). If vsync is set it waits for the vertical retrace, otherwise it sets the colors immediately.

See also:

See Section 11.4 [set_palette], page 94.

See Section 11.8 [get_palette_range], page 95.

11.6 get_color

`void get_color(int index, RGB *p);`

Retrieves the specified palette entry.

See also:

See Section 11.7 [get_palette], page 94.

See Section 11.2 [set_color], page 93.

11.7 get_palette

`void get_palette(PALETTE p);`

Retrieves the entire palette of 256 colors. You should provide an array of 256 RGB structures to store it in.

See also:

See Section 11.8 [get_palette_range], page 95.

See Section 11.6 [get_color], page 94.

See Section 11.4 [set_palette], page 94.

11.8 get_palette_range

`void get_palette_range(PALETTE p, int from, int to);`

Retrieves the palette entries between from and to (inclusive: pass 0 and 255 to get the entire palette).

See also:

See Section 11.7 [get_palette], page 94.

See Section 11.5 [set_palette_range], page 94.

11.9 fade_interpolate

`void fade_interpolate(const PALETTE source, const PALETTE dest, PALETTE output, int pos, int from, to);`

Calculates a temporary palette part way between source and dest, returning it in the output parameter. The position between the two extremes is specified by the pos value: 0 returns an exact copy of source, 64 returns dest, 32 returns a palette half way between the two, etc. This routine only affects colors between from and to (inclusive: pass 0 and 255 to interpolate the entire palette).

See also:

See Section 11.14 [fade_in], page 96.

See Section 11.15 [fade_out], page 97.

See Section 11.13 [fade_from], page 96.

11.10 fade_from_range

`void fade_from_range(const PALETTE source, const PALETTE dest, int speed, int from, to);`

Gradually fades a part of the palette from the source palette to the dest palette. The speed is from 1 (the slowest) up to 64 (instantaneous). This routine only affects colors between from and to (inclusive: pass 0 and 255 to fade the entire palette).

See also:

See Section 11.13 [fade_from], page 96.

11.11 fade_in_range

`void fade_in_range(const PALETTE p, int speed, int from, to);`

Gradually fades a part of the palette from a black screen to the specified palette. The speed is from 1 (the slowest) up to 64 (instantaneous). This routine only affects colors between from and to (inclusive: pass 0 and 255 to fade the entire palette).

See also:

See Section 11.14 [fade_in], page 96.

11.12 fade_out_range

`void fade_out_range(int speed, int from, to);`

Gradually fades a part of the palette from the current palette to a black screen. The speed is from 1 (the slowest) up to 64 (instantaneous). This routine only affects colors between from and to (inclusive: pass 0 and 255 to fade the entire palette).

See also:

See Section 11.15 [fade_out], page 97.

11.13 fade_from

`void fade_from(const PALETTE source, const PALETTE dest, int speed);`

Fades gradually from the source palette to the dest palette. The speed is from 1 (the slowest) up to 64 (instantaneous).

See also:

See Section 11.14 [fade_in], page 96.

See Section 11.15 [fade_out], page 97.

See Section 11.9 [fade_interpolate], page 95.

See Section 11.10 [fade_from_range], page 95.

11.14 fade_in

`void fade_in(const PALETTE p, int speed);`

Fades gradually from a black screen to the specified palette. The speed is from 1 (the slowest) up to 64 (instantaneous).

See also:

See Section 11.15 [fade_out], page 97.

See Section 11.13 [fade_from], page 96.

See Section 11.9 [fade_interpolate], page 95.

See Section 11.11 [fade_in_range], page 95.

11.15 fade_out

`void fade_out(int speed);`

Fades gradually from the current palette to a black screen. The speed is from 1 (the slowest) up to 64 (instantaneous).

See also:

See Section 11.14 [fade_in], page 96.

See Section 11.13 [fade_from], page 96.

See Section 11.9 [fade_interpolate], page 95.

See Section 11.11 [fade_in_range], page 95.

11.16 select_palette

`void select_palette(const PALETTE p);`

Ugly hack for use in various dodgy situations where you need to convert between paletted and truecolor image formats. Sets the internal palette table in the same way as the set_palette() function, so the conversion will use the specified palette, but without affecting the display hardware in any way. The previous palette settings are stored in an internal buffer, and can be restored by calling unselect_palette().

See also:

See Section 11.4 [set_palette], page 94.

See Section 11.17 [unselect_palette], page 97.

11.17 unselect_palette

`void unselect_palette();`

Restores the palette tables that were in use before the last call to select_palette().

See also:

See Section 11.16 [select_palette], page 97.

11.18 generate_332_palette

`void generate_332_palette(PALETTE pal);`

Constructs a fake truecolor palette, using three bits for red and green and two for the blue. The load_bitmap() function returns this if the file does not contain a palette itself (ie. you are reading a truecolor bitmap).

See also:

See Section 11.19 [generate_optimized_palette], page 98.

See Section 8.1 [set_color_depth], page 68.

11.19 generate_optimized_palette

`int generate_optimized_palette(BITMAP *bmp, PALETTE pal, const char rsvd[256]);`

Generates a 256 color palette suitable for making a reduced color version of the specified truecolor image. The rsvd parameter points to a table indicating which colors it is allowed to modify: zero for free colors which may be set to whatever the optimiser likes, negative values for reserved colors which cannot be used, and positive values for fixed palette entries that must not be changed, but can be used in the optimisation.

See also:

See Section 11.18 [generate_332_palette], page 97.

See Section 8.1 [set_color_depth], page 68.

11.20 default_palette

`extern PALETTE default_palette;`

The default IBM BIOS palette. This will be automatically selected whenever you set a new graphics mode.

See also:

See Section 11.21 [black_palette], page 98.

See Section 11.22 [desktop_palette], page 98.

11.21 black_palette

`extern PALETTE black_palette;`

A palette containing solid black colors, used by the fade routines.

See also:

See Section 11.20 [default_palette], page 98.

See Section 11.22 [desktop_palette], page 98.

11.22 desktop_palette

`extern PALETTE desktop_palette;`

The palette used by the Atari ST low resolution desktop. I'm not quite sure why this is still here, except that the grabber and test programs use it. It is probably the only Atari legacy code left in Allegro, and it would be a shame to remove it :-)

See also:
See Section 11.20 [default_palette], page 98.
See Section 11.21 [black_palette], page 98.

12 Truecolor pixel formats

In a truecolor video mode the red, green, and blue components for each pixel are packed directly into the color value, rather than using a palette lookup table. In a 15 bit mode there are 5 bits for each color, in 16 bit modes there are 5 bits each of red and blue and six bits of green, and both 24 and 32 bit modes use 8 bits for each color (the 32 bit pixels simply have an extra padding byte to align the data nicely). The layout of these components can vary depending on your hardware, but will generally either be RGB or BGR. Since the layout is not known until you select the video mode you will be using, you must call set_gfx_mode() before using any of the following routines!

12.1 makecol8

```
int makecol8(int r, int g, int b);
int makecol15(int r, int g, int b);
int makecol16(int r, int g, int b);
int makecol24(int r, int g, int b);
int makecol32(int r, int g, int b);
```

These functions convert colors from a hardware independent form (red, green, and blue values ranging 0-255) into various display dependent pixel formats. Converting to 15, 16, 24, or 32 bit formats only takes a few shifts, so it is fairly efficient. Converting to an 8 bit color involves searching the palette to find the closest match, which is quite slow unless you have set up an RGB mapping table (see below).

See also:
See Section 12.2 [makeacol32], page 99.
See Section 12.3 [makecol], page 100.
See Section 12.4 [makecol_depth], page 100.
See Section 12.6 [makecol15_dither], page 101.
See Section 20.2 [rgb_map], page 149.
See Section 20.1 [bestfit_color], page 149.
See Section 8.1 [set_color_depth], page 68.

12.2 makeacol32

```
int makeacol32(int r, int g, int b, int a);
```

Converts an RGBA color into a 32 bit display pixel format, which includes an alpha (transparency) value. There are no versions of this routine for other color depths, because only the 32 bit format has enough room to store a proper alpha channel. You should only use RGBA format colors as the input to draw_trans_sprite() or draw_trans_rle_sprite() after calling set_alpha_blender(), rather than drawing them directly to the screen.

See also:

See Section 12.5 [makeacol], page 100.

See Section 19.12 [set_alpha_blender], page 144.

See Section 19.13 [set_write_alpha_blender], page 144.

12.3 makecol

`int makecol(int r, int g, int b);`

Converts colors from a hardware independent format (red, green, and blue values ranging 0-255) to the pixel format required by the current video mode, calling the preceding 8, 15, 16, 24, or 32 bit makecol functions as appropriate.

See also:

See Section 12.5 [makeacol], page 100.

See Section 12.1 [makecol8], page 99.

See Section 12.4 [makecol_depth], page 100.

See Section 12.6 [makecol15_dither], page 101.

See Section 20.2 [rgb_map], page 149.

See Section 8.1 [set_color_depth], page 68.

12.4 makecol_depth

`int makecol_depth(int color_depth, int r, int g, int b);`

Converts colors from a hardware independent format (red, green, and blue values ranging 0-255) to the pixel format required by the specified color depth.

See also:

See Section 12.5 [makeacol], page 100.

See Section 12.3 [makecol], page 100.

See Section 12.1 [makecol8], page 99.

See Section 12.6 [makecol15_dither], page 101.

See Section 20.2 [rgb_map], page 149.

See Section 8.1 [set_color_depth], page 68.

12.5 makeacol

```
int makeacol(int r, int g, int b, int a);
```

```
int makeacol_depth(int color_depth, int r, int g, int b, int a);
```

Convert RGBA colors into display dependent pixel formats. In anything less than a 32 bit mode, these are the same as calling makecol() or makecol_depth(), but by using these routines it is possible to create 32 bit color values that contain a true 8 bit alpha channel along with the red, green, and blue components. You should only use RGBA format colors as the input to draw_trans_sprite() or draw_trans_rle_sprite() after calling set_alpha_blender(), rather than drawing them directly to the screen.

See also:

See Section 12.3 [makecol], page 100.

See Section 12.4 [makecol_depth], page 100.

See Section 19.12 [set_alpha_blender], page 144.

See Section 19.13 [set_write_alpha_blender], page 144.

12.6 makecol15_dither

```
int makecol15_dither(int r, int g, int b, int x, int y);
```

```
int makecol16_dither(int r, int g, int b, int x, int y);
```

Given both a color value and a pixel coordinate, calculate a dithered 15 or 16 bit RGB value. This can produce better results when reducing images from truecolor to hicolor. In addition to calling these functions directly, hicolor dithering can be automatically enabled when loading graphics by calling the set_color_conversion() function, for example set_color_conversion (COLORCONV_REDUCE_TRUE_TO_HI | COLORCONV_DITHER).

See also:

See Section 12.3 [makecol], page 100.

See Section 12.1 [makecol8], page 99.

See Section 10.11 [set_color_conversion], page 90.

12.7 getr8

```
int getr8(int c);
```
```
int getg8(int c);
```
```
int getb8(int c);
```
```
int getr15(int c);
```
```
int getg15(int c);
```
```
int getb15(int c);
```
```
int getr16(int c);
```
```
int getg16(int c);
```
```
int getb16(int c);
```
```
int getr24(int c);
```
```
int getg24(int c);
```
```
int getb24(int c);
```
```
int getr32(int c);
```
```
int getg32(int c);
```
```
int getb32(int c);
```

Given a color in a display dependent format, these functions extract one of the red, green, or blue components (ranging 0-255).

See also:
See Section 12.8 [geta32], page 102.
See Section 12.9 [getr], page 102.
See Section 12.10 [getr_depth], page 103.
See Section 12.3 [makecol], page 100.
See Section 8.1 [set_color_depth], page 68.

12.8 geta32

```
int geta32(int c);
```

Given a color in a 32 bit pixel format, this function extracts the alpha component (ranging 0-255).

See also:
See Section 12.7 [getr8], page 101.

12.9 getr

```
int getr(int c);
int getg(int c);
int getb(int c);
int geta(int c);
```

Given a color in the format being used by the current video mode, these functions extract one of the red, green, blue, or alpha components (ranging 0-255), calling the preceding 8, 15, 16, 24, or 32 bit get functions as appropriate. The alpha part is only meaningful for 32 bit pixels.

See also:
See Section 12.7 [getr8], page 101.
See Section 12.10 [getr_depth], page 103.
See Section 12.3 [makecol], page 100.
See Section 8.1 [set_color_depth], page 68.

12.10 getr_depth

```
int getr_depth(int color_depth, int c);
int getg_depth(int color_depth, int c);
int getb_depth(int color_depth, int c);
int geta_depth(int color_depth, int c);
```

Given a color in the format being used by the specified color depth, these functions extract one of the red, green, blue, or alpha components (ranging 0-255). The alpha part is only meaningful for 32 bit pixels.

See also:
See Section 12.9 [getr], page 102.
See Section 12.7 [getr8], page 101.
See Section 12.8 [geta32], page 102.
See Section 12.3 [makecol], page 100.
See Section 8.1 [set_color_depth], page 68.

12.11 palette_color

```
extern int palette_color[256];
```

Table mapping palette index colors (0-255) into whatever pixel format is being used by the current display mode. In a 256 color mode this just maps onto the array index. In truecolor modes it looks up the specified entry in the current palette, and converts that RGB value into the appropriate packed pixel format.

See also:
See Section 11.4 [set_palette], page 94.

See Section 12.3 [makecol], page 100.
See Section 8.1 [set_color_depth], page 68.

12.12 MASK_COLOR_8

```
#define MASK_COLOR_8 0
```

```
#define MASK_COLOR_15 (5.5.5 pink)
```

```
#define MASK_COLOR_16 (5.6.5 pink)
```

```
#define MASK_COLOR_24 (8.8.8 pink)
```

```
#define MASK_COLOR_32 (8.8.8 pink)
```

Constants representing the colors used to mask transparent sprite pixels for each color depth. In 256 color resolutions this is zero, and in truecolor modes it is bright pink (maximum red and blue, zero green).

See also:
See Section 9.10 [bitmap_mask_color], page 83.
See Section 12.3 [makecol], page 100.
See Section 14.8 [draw_sprite], page 114.
See Section 14.6 [masked_blit], page 113.

13 Drawing primitives

Except for _putpixel(), all these routines are affected by the current drawing mode and the clipping rectangle of the destination bitmap.

13.1 putpixel

```
void putpixel(BITMAP *bmp, int x, int y, int color);
```

Writes a pixel to the specified position in the bitmap, using the current drawing mode and the bitmap's clipping rectangle.

See also:
See Section 13.3 [getpixel], page 105.
See Section 13.2 [_putpixel], page 104.
See Section 19.1 [drawing_mode], page 138.

13.2 _putpixel

`void _putpixel(BITMAP *bmp, int x, int y, int color);`

`void _putpixel15(BITMAP *bmp, int x, int y, int color);`

`void _putpixel16(BITMAP *bmp, int x, int y, int color);`

`void _putpixel24(BITMAP *bmp, int x, int y, int color);`

`void _putpixel32(BITMAP *bmp, int x, int y, int color);`

Like the regular putpixel(), but much faster because they are implemented as an inline assembler functions for specific color depths. These won't work in mode-X graphics modes, don't perform any clipping (they will crash if you try to draw outside the bitmap!), and ignore the drawing mode.

See also:

See Section 13.1 [putpixel], page 104.

13.3 getpixel

`int getpixel(BITMAP *bmp, int x, int y);`

Reads a pixel from point x, y in the bitmap. Returns -1 if the point lies outside the bitmap.

See also:

See Section 13.1 [putpixel], page 104.

See Section 13.4 [_getpixel], page 105.

13.4 _getpixel

`int _getpixel(BITMAP *bmp, int x, int y);`

`int _getpixel15(BITMAP *bmp, int x, int y);`

`int _getpixel16(BITMAP *bmp, int x, int y);`

`int _getpixel24(BITMAP *bmp, int x, int y);`

`int _getpixel32(BITMAP *bmp, int x, int y);`

Faster inline versions of getpixel() for specific color depths. These won't work in mode-X, and don't do any clipping, so you must make sure the point lies inside the bitmap.

See also:

See Section 13.3 [getpixel], page 105.

13.5 vline

`void vline(BITMAP *bmp, int x, int y1, int y2, int color);`

Draws a vertical line onto the bitmap, from point (x, y1) to (x, y2).

See also:
See Section 13.6 [hline], page 106.
See Section 13.8 [line], page 106.
See Section 19.1 [drawing_mode], page 138.

13.6 hline

`void hline(BITMAP *bmp, int x1, int y, int x2, int color);`

Draws a horizontal line onto the bitmap, from point (x1, y) to (x2, y).

See also:
See Section 13.5 [vline], page 105.
See Section 13.8 [line], page 106.
See Section 19.1 [drawing_mode], page 138.

13.7 do_line

`void do_line(BITMAP *bmp, int x1, y1, x2, y2, int d, void (*proc)(BITMAP *bmp, int x, int y, int d));`

Calculates all the points along a line from point (x1, y1) to (x2, y2), calling the supplied function for each one. This will be passed a copy of the bmp parameter, the x and y position, and a copy of the d parameter, so it is suitable for use with putpixel().

See also:
See Section 13.13 [do_circle], page 108.
See Section 13.16 [do_ellipse], page 108.
See Section 13.19 [do_arc], page 109.
See Section 13.8 [line], page 106.

13.8 line

`void line(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);`

Draws a line onto the bitmap, from point (x1, y1) to (x2, y2).

See also:
See Section 13.6 [hline], page 106.
See Section 13.5 [vline], page 105.
See Section 13.7 [do_line], page 106.
See Section 19.1 [drawing_mode], page 138.

13.9 triangle

```
void triangle(BITMAP *bmp, int x1, y1, x2, y2, x3, y3, int color);
```

Draws a filled triangle between the three points.

See also:

See Section 13.10 [polygon], page 107.

See Section 18.2 [triangle3d], page 130.

See Section 19.1 [drawing_mode], page 138.

13.10 polygon

```
void polygon(BITMAP *bmp, int vertices, const int *points, int color);
```

Draws a filled polygon with an arbitrary number of corners. Pass the number of vertices and an array containing a series of x, y points (a total of vertices*2 values).

See also:

See Section 13.9 [triangle], page 107.

See Section 18.1 [polygon3d], page 127.

See Section 19.1 [drawing_mode], page 138.

13.11 rect

```
void rect(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);
```

Draws an outline rectangle with the two points as its opposite corners.

See also:

See Section 13.12 [rectfill], page 107.

See Section 19.1 [drawing_mode], page 138.

13.12 rectfill

```
void rectfill(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);
```

Draws a solid, filled rectangle with the two points as its opposite corners.

See also:

See Section 13.11 [rect], page 107.

See Section 14.1 [clear_bitmap], page 111.

See Section 19.1 [drawing_mode], page 138.

13.13 do_circle

`void do_circle(BITMAP *bmp, int x, int y, int radius, int d, void (*proc)(BITMAP *bmp, int x, int y, int d));`

Calculates all the points in a circle around point (x, y) with radius r, calling the supplied function for each one. This will be passed a copy of the bmp parameter, the x and y position, and a copy of the d parameter, so it is suitable for use with putpixel().

See also:

See Section 13.16 [do_ellipse], page 108.

See Section 13.19 [do_arc], page 109.

See Section 13.7 [do_line], page 106.

See Section 13.14 [circle], page 108.

See Section 13.15 [circlefill], page 108.

13.14 circle

`void circle(BITMAP *bmp, int x, int y, int radius, int color);`

Draws a circle with the specified centre and radius.

See also:

See Section 13.17 [ellipse], page 109.

See Section 13.20 [arc], page 110.

See Section 13.15 [circlefill], page 108.

See Section 13.13 [do_circle], page 108.

See Section 19.1 [drawing_mode], page 138.

13.15 circlefill

`void circlefill(BITMAP *bmp, int x, int y, int radius, int color);`

Draws a filled circle with the specified centre and radius.

See also:

See Section 13.18 [ellipsefill], page 109.

See Section 13.14 [circle], page 108.

See Section 13.13 [do_circle], page 108.

See Section 19.1 [drawing_mode], page 138.

13.16 do_ellipse

```
void do_ellipse(BITMAP *bmp, int x, int y, int rx, ry, int d, void
(*proc)(BITMAP *bmp, int x, int y, int d));
```

Calculates all the points in an ellipse around point (x, y) with radius rx and ry, calling the supplied function for each one. This will be passed a copy of the bmp parameter, the x and y position, and a copy of the d parameter, so it is suitable for use with putpixel().

See also:
See Section 13.13 [do_circle], page 108.
See Section 13.19 [do_arc], page 109.
See Section 13.7 [do_line], page 106.
See Section 13.17 [ellipse], page 109.
See Section 13.18 [ellipsefill], page 109.

13.17 ellipse

```
void ellipse(BITMAP *bmp, int x, int y, int rx, int ry, int color);
```

Draws an ellipse with the specified centre and radius.

See also:
See Section 13.14 [circle], page 108.
See Section 13.20 [arc], page 110.
See Section 13.18 [ellipsefill], page 109.
See Section 13.16 [do_ellipse], page 108.
See Section 19.1 [drawing_mode], page 138.

13.18 ellipsefill

```
void ellipsefill(BITMAP *bmp, int x, int y, int rx, int ry, int color);
```

Draws a filled ellipse with the specified centre and radius.

See also:
See Section 13.15 [circlefill], page 108.
See Section 13.17 [ellipse], page 109.
See Section 13.16 [do_ellipse], page 108.
See Section 19.1 [drawing_mode], page 138.

13.19 do_arc

```
void do_arc(BITMAP *bmp, int x, int y, fixed a1, fixed a2, int r, int d,
void (*proc)(BITMAP *bmp, int x, int y, int d));
```

Calculates all the points in a circular arc around point (x, y) with radius r, calling the supplied function for each one. This will be passed a copy of the bmp parameter, the x and y position, and a copy of the d parameter, so it is suitable for use with putpixel(). The arc will be plotted in an anticlockwise direction starting from the angle a1 and ending when it reaches a2. These values are specified in 16.16 fixed point format, with 256 equal to a full circle, 64 a right angle, etc. Zero is to the right of the centre point, and larger values rotate anticlockwise from there.

See also:

See Section 13.13 [do_circle], page 108.

See Section 13.16 [do_ellipse], page 108.

See Section 13.7 [do_line], page 106.

See Section 13.20 [arc], page 110.

13.20 arc

```
void arc(BITMAP *bmp, int x, y, fixed ang1, ang2, int r, int color);
```

Draws a circular arc with centre x, y and radius r, in an anticlockwise direction starting from the angle a1 and ending when it reaches a2. These values are specified in 16.16 fixed point format, with 256 equal to a full circle, 64 a right angle, etc. Zero is to the right of the centre point, and larger values rotate anticlockwise from there.

See also:

See Section 13.14 [circle], page 108.

See Section 13.17 [ellipse], page 109.

See Section 19.1 [drawing_mode], page 138.

13.21 calc_spline

```
void calc_spline(const int points[8], int npts, int *x, int *y);
```

Calculates a series of npts values along a bezier spline, storing them in the output x and y arrays. The bezier curve is specified by the four x/y control points in the points array: points[0] and points[1] contain the coordinates of the first control point, points[2] and points[3] are the second point, etc. Control points 0 and 3 are the ends of the spline, and points 1 and 2 are guides. The curve probably won't pass through points 1 and 2, but they affect the shape of the curve between points 0 and 3 (the lines p0-p1 and p2-p3 are tangents to the spline). The easiest way to think of it is that the curve starts at p0, heading in the direction of p1, but curves round so that it arrives at p3 from the direction of p2. In addition to their role as graphics primitives, spline curves can be useful for constructing smooth paths around a series of control points, as in exspline.c.

See also:

See Section 13.22 [spline], page 111.

13.22 spline

```
void spline(BITMAP *bmp, const int points[8], int color);
```

Draws a bezier spline using the four control points specified in the points array.

See also:

See Section 13.21 [calc_spline], page 110.

See Section 19.1 [drawing_mode], page 138.

13.23 floodfill

```
void floodfill(BITMAP *bmp, int x, int y, int color);
```

Floodfills an enclosed area, starting at point (x, y), with the specified color.

See also:

See Section 19.1 [drawing_mode], page 138.

14 Blitting and sprites

All these routines are affected by the clipping rectangle of the destination bitmap.

14.1 clear_bitmap

```
void clear_bitmap(BITMAP *bitmap);
```

Clears the bitmap to color 0.

See also:

See Section 14.3 [clear_to_color], page 112.

See Section 14.2 [clear], page 111.

14.2 clear

```
void clear(BITMAP *bitmap);
```

An alias for clear_bitmap(), provided for backwards compatibility. It is implemented as a static inline function. The aliasing may be switched off by defining the preprocessor symbol ALLEGRO_NO_CLEAR_BITMAP_ALIAS before including Allegro headers, eg:

```
#define ALLEGRO_NO_CLEAR_BITMAP_ALIAS
#include <allegro.h>
```

See also:

See Section 14.1 [clear_bitmap], page 111.

14.3 clear_to_color

`void clear_to_color(BITMAP *bitmap, int color);`

Clears the bitmap to the specified color.

See also:

See Section 14.1 [clear_bitmap], page 111.

14.4 blit

`void blit(BITMAP *source, BITMAP *dest, int source_x, int source_y, int dest_x, int dest_y, int width, int height);`

Copies a rectangular area of the source bitmap to the destination bitmap. The source_x and source_y parameters are the top left corner of the area to copy from the source bitmap, and dest_x and dest_y are the corresponding position in the destination bitmap. This routine respects the destination clipping rectangle, and it will also clip if you try to blit from areas outside the source bitmap.

You can blit between any parts of any two bitmaps, even if the two memory areas overlap (ie. source and dest are the same, or one is sub-bitmap of the other). You should be aware, however, that a lot of SVGA cards don't provide separate read and write banks, which means that blitting from one part of the screen to another requires the use of a temporary bitmap in memory, and is therefore extremely slow. As a general rule you should avoid blitting from the screen onto itself in SVGA modes.

In mode-X, on the other hand, blitting from one part of the screen to another can be significantly faster than blitting from memory onto the screen, as long as the source and destination are correctly aligned with each other. Copying between overlapping screen rectangles is slow, but if the areas don't overlap, and if they have the same plane alignment (ie. (source_x%4) == (dest_x%4)), the VGA latch registers can be used for a very fast data transfer. To take advantage of this, in mode-X it is often worth storing tile graphics in a hidden area of video memory (using a large virtual screen), and blitting them from there onto the visible part of the screen.

If the GFX_HW_VRAM_BLIT bit in the gfx_capabilities flag is set, the current driver supports hardware accelerated blits from one part of the screen onto another. This is extremely fast, so when this flag is set it may be worth storing some of your more frequently used graphics in an offscreen portion of the video memory.

Unlike most of the graphics routines, blit() allows the source and destination bitmaps to be of different color depths, so it can be used to convert images from one pixel format to another.

See also:
See Section 14.6 [masked_blit], page 113.
See Section 14.5 [stretch_blit], page 113.
See Section 14.8 [draw_sprite], page 114.
See Section 8.11 [gfx_capabilities], page 74.

14.5 stretch_blit

`void stretch_blit(BITMAP *source, BITMAP *dest, int source_x, source_y, source_width, source_height, int dest_x, dest_y, dest_width, dest_height);`

Like blit(), except it can scale images (so the source and destination rectangles don't need to be the same size) and requires the source and destination bitmaps to be of the same color depth. This routine doesn't do as much safety checking as the regular blit(): in particular you must take care not to copy from areas outside the source bitmap, and you cannot blit between overlapping regions, ie. you must use different bitmaps for the source and the destination. Moreover, the source must be a memory bitmap.

See also:
See Section 14.4 [blit], page 112.
See Section 14.7 [masked_stretch_blit], page 114.
See Section 14.9 [stretch_sprite], page 115.

14.6 masked_blit

`void masked_blit(BITMAP *source, BITMAP *dest, int source_x, int source_y, int dest_x, int dest_y, int width, int height);`

Like blit(), but skips transparent pixels, which are marked by a zero in 256 color modes or bright pink for truecolor data (maximum red and blue, zero green), and requires the source and destination bitmaps to be of the same color depth. The source and destination regions must not overlap.

If the GFX_HW_VRAM_BLIT_MASKED bit in the gfx_capabilities flag is set, the current driver supports hardware accelerated masked blits from one part of the screen onto another. This is extremely fast, so when this flag is set it may be worth storing some of your more frequently used sprites in an offscreen portion of the video memory.

Warning: if the hardware acceleration flag is not set, masked_blit() will not work correctly when used with a source image in system or video memory so the latter must be a memory bitmap.

See also:
See Section 14.4 [blit], page 112.
See Section 14.7 [masked_stretch_blit], page 114.
See Section 14.8 [draw_sprite], page 114.
See Section 9.10 [bitmap_mask_color], page 83.

14.7 masked_stretch_blit

```
void masked_stretch_blit(BITMAP *source, BITMAP *dest, int source_x,
source_y, source_w, source_h, int dest_x, dest_y, dest_w, dest_h);
```

Like masked_blit(), except it can scale images (so the source and destination rectangles don't need to be the same size). This routine doesn't do as much safety checking as the regular masked_blit(): in particular you must take care not to copy from areas outside the source bitmap. Moreover, the source must be a memory bitmap.

See also:
See Section 14.4 [blit], page 112.
See Section 14.6 [masked_blit], page 113.
See Section 14.5 [stretch_blit], page 113.
See Section 14.9 [stretch_sprite], page 115.

14.8 draw_sprite

```
void draw_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y);
```

Draws a copy of the sprite bitmap onto the destination bitmap at the specified position. This is almost the same as blit(sprite, bmp, 0, 0, x, y, sprite->w, sprite->h), but it uses a masked drawing mode where transparent pixels are skipped, so the background image will show through the masked parts of the sprite. Transparent pixels are marked by a zero in 256 color modes or bright pink for truecolor data (maximum red and blue, zero green).

If the GFX_HW_VRAM_BLIT_MASKED bit in the gfx_capabilities flag is set, the current driver supports hardware accelerated sprite drawing when the source image is a video memory bitmap or a sub-bitmap of the screen. This is extremely fast, so when this flag is set it may be worth storing some of your more frequently used sprites in an offscreen portion of the video memory.

Warning: if the hardware acceleration flag is not set, draw_sprite() will not work correctly when used with a sprite image in system or video memory so the latter must be a memory bitmap.

Although generally not supporting graphics of mixed color depths, as a special case this function can be used to draw 256 color source images onto truecolor destination bitmaps, so you can use palette effects on specific sprites within a truecolor program.

See also:

See Section 14.10 [draw_sprite_v_flip], page 115.

See Section 14.11 [draw_trans_sprite], page 116.

See Section 14.12 [draw_lit_sprite], page 116.

See Section 14.13 [draw_gouraud_sprite], page 117.

See Section 14.9 [stretch_sprite], page 115.

See Section 14.15 [rotate_sprite], page 117.

See Section 14.14 [draw_character], page 117.

See Section 15.3 [draw_rle_sprite], page 121.

See Section 16.3 [draw_compiled_sprite], page 123.

See Section 14.6 [masked_blit], page 113.

See Section 14.4 [blit], page 112.

See Section 9.10 [bitmap_mask_color], page 83.

14.9 stretch_sprite

```
void stretch_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y, int w, int
h);
```

Like draw_sprite(), except it can stretch the sprite image to the specified width and height and requires the sprite image and destination bitmap to be of the same color depth. Moreover, the sprite image must be a memory bitmap.

See also:

See Section 14.8 [draw_sprite], page 114.

See Section 14.5 [stretch_blit], page 113.

See Section 9.10 [bitmap_mask_color], page 83.

14.10 draw_sprite_v_flip

```
void draw_sprite_v_flip(BITMAP *bmp, BITMAP *sprite, int x, int y);
```

```
void draw_sprite_h_flip(BITMAP *bmp, BITMAP *sprite, int x, int y);
```

```
void draw_sprite_vh_flip(BITMAP *bmp, BITMAP *sprite, int x, int y);
```

These are like draw_sprite(), but they flip the image about the vertical, horizontal, or both, axes. This produces exact mirror images, which is not the same as rotating the sprite (and it is a lot faster than the rotation routine). The sprite must be a memory bitmap.

See also:

See Section 14.8 [draw_sprite], page 114.

See Section 9.10 [bitmap_mask_color], page 83.

14.11 draw_trans_sprite

`void draw_trans_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y);`

Uses the global color_map table or truecolor blender functions to overlay the sprite on top of the existing image. This must only be used after you have set up the color mapping table (for 256 color modes) or blender functions (for truecolor modes). Because it involves reading as well as writing the bitmap memory, translucent drawing is very slow if you draw directly to video RAM, so wherever possible you should use a memory bitmap instead. The bitmap and sprite must normally be in the same color depth, but as a special case you can draw 32 bit RGBA format sprites onto any hicolor or truecolor bitmap, as long as you call set_alpha_blender() first, and you can draw 8 bit alpha images onto a 32 bit RGBA destination, as long as you call set_write_alpha_blender() first.

See also:

See Section 14.8 [draw_sprite], page 114.

See Section 14.12 [draw_lit_sprite], page 116.

See Section 15.4 [draw_trans_rle_sprite], page 121.

See Section 19.5 [color_map], page 140.

See Section 19.11 [set_trans_blender], page 143.

See Section 19.12 [set_alpha_blender], page 144.

See Section 19.13 [set_write_alpha_blender], page 144.

See Section 9.10 [bitmap_mask_color], page 83.

14.12 draw_lit_sprite

`void draw_lit_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y, int color);`

In 256 color modes, uses the global color_map table to tint the sprite image to the specified color or to light it to the level specified by 'color', depending on the function which was used to build the table (create_trans_table or create_light_table), and draws the resulting image to the destination bitmap. In truecolor modes, uses the blender functions to light the sprite image using the alpha level specified by 'color' (the alpha level which was passed to the blender functions is ignored) and draws the resulting image to the destination bitmap. The 'color' parameter must be in the range [0-255] whatever its actual meaning is. This must only be used after you have set up the color mapping table (for 256 color modes) or blender functions (for truecolor modes).

See also:

See Section 14.8 [draw_sprite], page 114.

See Section 14.11 [draw_trans_sprite], page 116.

See Section 14.13 [draw_gouraud_sprite], page 117.

See Section 15.5 [draw_lit_rle_sprite], page 122.

See Section 19.5 [color_map], page 140.

See Section 19.11 [set_trans_blender], page 143.

See Section 9.10 [bitmap_mask_color], page 83.

14.13 draw_gouraud_sprite

`void draw_gouraud_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y, int c1, int c2, int c3, int c4);`

More sophisticated version of draw_lit_sprite(): the 'color' parameter is not constant across the sprite image anymore but interpolated between the four specified corner colors, which have the same actual meaning as it.

See also:

See Section 14.8 [draw_sprite], page 114.

See Section 14.12 [draw_lit_sprite], page 116.

See Section 19.5 [color_map], page 140.

See Section 19.11 [set_trans_blender], page 143.

See Section 9.10 [bitmap_mask_color], page 83.

14.14 draw_character

`void draw_character(BITMAP *bmp, BITMAP *sprite, int x, int y, int color);`

Draws a copy of the sprite bitmap onto the destination bitmap at the specified position, drawing transparent pixels in the current text mode (skipping them if the text mode is -1, otherwise drawing them in the text background color), and setting all other pixels to the specified color. Transparent pixels are marked by a zero in 256 color modes or bright pink for truecolor data (maximum red and blue, zero green). The sprite must be an 8 bit image, even if the destination is a truecolor bitmap.

See also:

See Section 14.8 [draw_sprite], page 114.

See Section 9.10 [bitmap_mask_color], page 83.

14.15 rotate_sprite

`void rotate_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y, fixed angle);`

Draws the sprite image onto the bitmap. It is placed with its top left corner at the specified position, then rotated by the specified angle around its centre. The angle is a fixed point 16.16 number in the same format used by the fixed point trig routines, with 256 equal to a full circle, 64 a right angle, etc. All rotation functions can draw between any two bitmaps, even screen bitmaps or bitmaps of different color depth.

See also:

See Section 14.8 [draw_sprite], page 114.

See Section 14.17 [rotate_scaled_sprite], page 118.

See Section 14.16 [rotate_sprite_v_flip], page 118.

See Section 14.18 [rotate_scaled_sprite_v_flip], page 118.

See Section 14.19 [pivot_sprite], page 119.

See Section 14.20 [pivot_sprite_v_flip], page 119.

See Section 14.21 [pivot_scaled_sprite], page 119.

See Section 14.22 [pivot_scaled_sprite_v_flip], page 120.

See Section 30.1 [itofix], page 195.

See Section 30.11 [fixed point trig], page 197.

14.16 rotate_sprite_v_flip

```
void rotate_sprite_v_flip(BITMAP *bmp, BITMAP *sprite, int x, int y, fixed
angle);
```

Like rotate_sprite, but also flips the image vertically. To flip horizontally, use this routine but add itofix(128) to the angle. To flip in both directions, use rotate_sprite() and add itofix(128) to its angle.

See also:

See Section 14.15 [rotate_sprite], page 117.

See Section 14.18 [rotate_scaled_sprite_v_flip], page 118.

See Section 14.20 [pivot_sprite_v_flip], page 119.

See Section 14.22 [pivot_scaled_sprite_v_flip], page 120.

14.17 rotate_scaled_sprite

```
void rotate_scaled_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y, fixed
angle, fixed scale);
```

Like rotate_sprite(), but stretches or shrinks the image at the same time as rotating it.

See also:

See Section 14.15 [rotate_sprite], page 117.

See Section 14.18 [rotate_scaled_sprite_v_flip], page 118.

See Section 14.21 [pivot_scaled_sprite], page 119.

See Section 14.22 [pivot_scaled_sprite_v_flip], page 120.

14.18 rotate_scaled_sprite_v_flip

`void rotate_scaled_sprite_v_flip(BITMAP *bmp, BITMAP *sprite, int x, int y, fixed angle, fixed scale)`

Draws the sprite, similar to rotate_scaled_sprite() except that it flips the sprite vertically first.

See also:

See Section 14.15 [rotate_sprite], page 117.

See Section 14.17 [rotate_scaled_sprite], page 118.

See Section 14.16 [rotate_sprite_v_flip], page 118.

14.19 pivot_sprite

`void pivot_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y, int cx, int cy, fixed angle);`

Like rotate_sprite(), but aligns the point in the sprite given by (cx, cy) to (x, y) in the bitmap, then rotates around this point.

See also:

See Section 14.15 [rotate_sprite], page 117.

See Section 14.21 [pivot_scaled_sprite], page 119.

See Section 14.20 [pivot_sprite_v_flip], page 119.

14.20 pivot_sprite_v_flip

`void pivot_sprite_v_flip(BITMAP *bmp, BITMAP *sprite, int x, int y, int cx, int cy, fixed angle);`

Like rotate_sprite_v_flip(), but aligns the point in the sprite given by (cx, cy) to (x, y) in the bitmap, then rotates around this point.

See also:

See Section 14.15 [rotate_sprite], page 117.

See Section 14.16 [rotate_sprite_v_flip], page 118.

See Section 14.19 [pivot_sprite], page 119.

14.21 pivot_scaled_sprite

`void pivot_scaled_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y, int cx, int cy, fixed angle, fixed scale));`

Like rotate_scaled_sprite(), but aligns the point in the sprite given by (cx, cy) to (x, y) in the bitmap, then rotates and scales around this point.

See also:

See Section 14.15 [rotate_sprite], page 117.

See Section 14.17 [rotate_scaled_sprite], page 118.

See Section 14.19 [pivot_sprite], page 119.

See Section 14.22 [pivot_scaled_sprite_v_flip], page 120.

14.22 pivot_scaled_sprite_v_flip

```
void pivot_scaled_sprite_v_flip(BITMAP *bmp, BITMAP *sprite, int x, int y,
fixed angle, fixed scale)
```

Like rotate_scaled_sprite_v_flip(), but aligns the point in the sprite given by (cx, cy) to (x, y) in the bitmap, then rotates and scales around this point.

See also:
See Section 14.15 [rotate_sprite], page 117.
See Section 14.18 [rotate_scaled_sprite_v_flip], page 118.
See Section 14.16 [rotate_sprite_v_flip], page 118.
See Section 14.19 [pivot_sprite], page 119.
See Section 14.21 [pivot_scaled_sprite], page 119.

15 RLE sprites

Because bitmaps can be used in so many different ways, the bitmap structure is quite complicated, and it contains a lot of data. In many situations, though, you will find yourself storing images that are only ever copied to the screen, rather than being drawn onto or used as filling patterns, etc. If this is the case you may be better off storing your images in RLE_SPRITE or COMPILED_SPRITE structures rather than regular bitmaps.

RLE sprites store the image in a simple run-length encoded format, where repeated zero pixels are replaced by a single length count, and strings of non-zero pixels are preceded by a counter giving the length of the solid run. RLE sprites are usually much smaller then normal bitmaps, both because of the run length compression, and because they avoid most of the overhead of the bitmap structure. They are often also faster than normal bitmaps, because rather than having to compare every single pixel with zero to determine whether it should be drawn, it is possible to skip over a whole run of zeros with a single add, or to copy a long run of non-zero pixels with fast string instructions.

Every silver lining has a cloud, though, and in the case of RLE sprites it is a lack of flexibility. You can't draw onto them, and you can't flip them, rotate them, or stretch them. In fact the only thing you can do with them is to blast them onto a bitmap with the draw_rle_sprite() function, which is equivalent to using draw_sprite() with a regular bitmap. You can convert bitmaps into RLE sprites at runtime, or you can create RLE sprite structures in grabber datafiles by making a new object of type 'RLE sprite'.

15.1 get_rle_sprite

```
RLE_SPRITE *get_rle_sprite(BITMAP *bitmap);
```

Creates an RLE sprite based on the specified bitmap (which must be a memory bitmap).

See also:
See Section 15.3 [draw_rle_sprite], page 121.
See Section 15.2 [destroy_rle_sprite], page 121.

15.2 destroy_rle_sprite

`void destroy_rle_sprite(RLE_SPRITE *sprite);`

Destroys an RLE sprite structure previously returned by get_rle_sprite().

See also:
See Section 15.1 [get_rle_sprite], page 120.

15.3 draw_rle_sprite

`void draw_rle_sprite(BITMAP *bmp, const RLE_SPRITE *sprite, int x, int y);`

Draws an RLE sprite onto a bitmap at the specified position.

See also:
See Section 15.1 [get_rle_sprite], page 120.
See Section 14.8 [draw_sprite], page 114.
See Section 16.3 [draw_compiled_sprite], page 123.
See Section 15.4 [draw_trans_rle_sprite], page 121.
See Section 15.5 [draw_lit_rle_sprite], page 122.
See Section 9.10 [bitmap_mask_color], page 83.

15.4 draw_trans_rle_sprite

```
void draw_trans_rle_sprite(BITMAP *bmp, const RLE_SPRITE *sprite, int x,
int y);
```

Translucent version of draw_rle_sprite(). See the description of draw_trans_sprite(). This must only be used after you have set up the color mapping table (for 256 color modes) or blender functions (for truecolor modes). The bitmap and sprite must normally be in the same color depth, but as a special case you can draw 32 bit RGBA format sprites onto any hicolor or truecolor bitmap, as long as you call set_alpha_blender() first.

See also:
See Section 15.3 [draw_rle_sprite], page 121.
See Section 15.5 [draw_lit_rle_sprite], page 122.
See Section 14.11 [draw_trans_sprite], page 116.
See Section 19.5 [color_map], page 140.
See Section 19.11 [set_trans_blender], page 143.

See Section 19.12 [set_alpha_blender], page 144.

See Section 9.10 [bitmap_mask_color], page 83.

15.5 draw_lit_rle_sprite

`void draw_lit_rle_sprite(BITMAP *bmp, const RLE_SPRITE *sprite, int x, y, color);`

Tinted version of draw_rle_sprite(). See the description of draw_lit_sprite(). This must only be used after you have set up the color mapping table (for 256 color modes) or blender functions (for truecolor modes).

See also:
See Section 15.3 [draw_rle_sprite], page 121.
See Section 15.4 [draw_trans_rle_sprite], page 121.
See Section 14.12 [draw_lit_sprite], page 116.
See Section 19.5 [color_map], page 140.
See Section 19.11 [set_trans_blender], page 143.
See Section 9.10 [bitmap_mask_color], page 83.

16 Compiled sprites

Compiled sprites are stored as actual machine code instructions that draw a specific image onto a bitmap, using mov instructions with immediate data values. This is the fastest way to draw a masked image: on my machine drawing compiled sprites is about five times as fast as using draw_sprite() with a regular bitmap. Compiled sprites are big, so if memory is tight you should use RLE sprites instead, and what you can do with them is even more restricted than with RLE sprites, because they don't support clipping. If you try to draw one off the edge of a bitmap, you will corrupt memory and probably crash the system. You can convert bitmaps into compiled sprites at runtime, or you can create compiled sprite structures in grabber datafiles by making a new object of type 'Compiled sprite' or 'Compiled x-sprite'.

16.1 get_compiled_sprite

`COMPILED_SPRITE *get_compiled_sprite(BITMAP *bitmap, int planar);`

Creates a compiled sprite based on the specified bitmap (which must be a memory bitmap). Compiled sprites are device-dependent, so you have to specify whether to compile it into a linear or planar format. Pass FALSE as the second parameter if you are going to be drawing it onto memory bitmaps or mode 13h and SVGA screen bitmaps, and pass TRUE if you are going to draw it onto mode-X or Xtended mode screen bitmaps.

See also:

See Section 16.3 [draw_compiled_sprite], page 123.

See Section 16.2 [destroy_compiled_sprite], page 123.

16.2 destroy_compiled_sprite

`void destroy_compiled_sprite(COMPILED_SPRITE *sprite);`

Destroys a compiled sprite structure previously returned by get_compiled_sprite().

See also:

See Section 16.1 [get_compiled_sprite], page 122.

16.3 draw_compiled_sprite

`void draw_compiled_sprite(BITMAP *bmp, const COMPILED_SPRITE *sprite, int x, int y);`

Draws a compiled sprite onto a bitmap at the specified position. The sprite must have been compiled for the correct type of bitmap (linear or planar). This function does not support clipping.

Hint: if not being able to clip compiled sprites is a problem, a neat trick is to set up a work surface (memory bitmap, mode-X virtual screen, or whatever) a bit bigger than you really need, and use the middle of it as your screen. That way you can draw slightly off the edge without any trouble...

See also:
See Section 16.1 [get_compiled_sprite], page 122.
See Section 14.8 [draw_sprite], page 114.
See Section 15.3 [draw_rle_sprite], page 121.
See Section 9.10 [bitmap_mask_color], page 83.

17 Text output

Allegro provides text output routines that work with both monochrome and color fonts, which can contain any number of Unicode character ranges. The grabber program can create fonts from sets of characters drawn in a bitmap file (see grabber.txt for more information), and can also import GRX or BIOS format font files. The font structure contains a number of hooks that can be used to extend it with your own custom drawing code: see the definition in allegro/text.h for details.

17.1 font

`extern FONT *font;`

A simple 8x8 fixed size font (the mode 13h BIOS default). If you want to alter the font used by the GUI routines, change this to point to one of your own fonts. This font contains the standard ASCII (U+20 to U+7F), Latin-1 (U+A1 to U+FF), and Latin Extended-A (U+0100 to U+017F) character ranges.

See also:

See Section 17.4 [textout], page 124.

See Section 17.8 [textprintf], page 125.

17.2 allegro_404_char

`extern int allegro_404_char;`

When Allegro cannot find a glyph it needs in a font, it will instead output the character given in allegro_404_char. By default, this is set to the caret symbol, '^'.

See also:

See Section 17.1 [font], page 123.

17.3 text_mode

`int text_mode(int mode);`

Sets the mode in which text will be drawn. Returns previous mode. If mode is zero or positive, text output will be opaque and the background of the characters will be set to color #mode. If mode is negative, text will be drawn transparently (ie. the background of the characters will not be altered). The default is a mode of zero.

See also:

See Section 17.4 [textout], page 124.

See Section 17.8 [textprintf], page 125.

17.4 textout

`void textout(BITMAP *bmp, const FONT *f, const char *s, int x, y, int color);`

Writes the string s onto the bitmap at position x, y, using the current text mode and the specified font and foreground color. If the color is -1 and a color font is in use, it will be drawn using the colors from the original font bitmap (the one you imported into the grabber program), which allows multicolored text output.

See also:

See Section 17.1 [font], page 123.

See Section 17.3 [text_mode], page 124.

See Section 17.5 [textout_centre], page 125.

See Section 17.6 [textout_right], page 125.

See Section 17.7 [textout_justify], page 125.

See Section 17.8 [textprintf], page 125.

See Section 17.13 [text_height], page 127.

See Section 17.12 [text_length], page 126.

17.5 textout_centre

`void textout_centre(BITMAP *bmp, const FONT *f, const char *s, int x, y, color);`

Like textout(), but interprets the x coordinate as the centre rather than the left edge of the string.

See also:

See Section 17.4 [textout], page 124.

See Section 17.9 [textprintf_centre], page 126.

17.6 textout_right

`void textout_right(BITMAP *bmp, const FONT *f, const char *s, int x, y, color);`

Like textout(), but interprets the x coordinate as the right rather than the left edge of the string.

See also:

See Section 17.4 [textout], page 124.

See Section 17.10 [textprintf_right], page 126.

17.7 textout_justify

`void textout_justify(BITMAP *bmp, const FONT *f, const char *s, int x1, int x2, int y, int diff, int color);`

Draws justified text within the region x1-x2. If the amount of spare space is greater than the diff value, it will give up and draw regular left justified text instead.

See also:

See Section 17.4 [textout], page 124.

See Section 17.11 [textprintf_justify], page 126.

17.8 textprintf

`void textprintf(BITMAP *bmp, const FONT *f, int x, y, color, const char *fmt, ...);`

Formatted text output, using a printf() style format string.

See also:

See Section 17.1 [font], page 123.

See Section 17.3 [text_mode], page 124.
See Section 17.9 [textprintf_centre], page 126.
See Section 17.10 [textprintf_right], page 126.
See Section 17.11 [textprintf_justify], page 126.
See Section 17.4 [textout], page 124.
See Section 17.13 [text_height], page 127.
See Section 17.12 [text_length], page 126.

17.9 textprintf_centre

```
void textprintf_centre(BITMAP *bmp, const FONT *f, int x, y, color, const
char *fmt, ...);
```

Like textprintf(), but interprets the x coordinate as the centre rather than the left edge of the string.

See also:
See Section 17.8 [textprintf], page 125.
See Section 17.5 [textout_centre], page 125.

17.10 textprintf_right

```
void textprintf_right(BITMAP *bmp, const FONT *f, int x, y, color, const
char *fmt, ...);
```

Like textprintf(), but interprets the x coordinate as the right rather than the left edge of the string.

See also:
See Section 17.8 [textprintf], page 125.
See Section 17.6 [textout_right], page 125.

17.11 textprintf_justify

```
void textprintf_justify(BITMAP *bmp, const FONT *f, int x1, int x2, int y,
int diff, int color, const char *fmt, ...);
```

Like textout_justify, but using a printf() style format string.

See also:
See Section 17.8 [textprintf], page 125.
See Section 17.7 [textout_justify], page 125.

17.12 text_length

`int text_length(const FONT *f, const char *str);`

Returns the length (in pixels) of a string in the specified font.

See also:

See Section 17.13 [text_height], page 127.

17.13 text_height

`int text_height(const FONT *f)`

Returns the height (in pixels) of the specified font.

See also:

See Section 17.12 [text_length], page 126.

17.14 destroy_font

`void destroy_font(FONT *f);`

Frees the memory being used by a font structure.

18 Polygon rendering

18.1 polygon3d

`void polygon3d(BITMAP *bmp, int type, BITMAP *texture, int vc, V3D *vtx[]);`

`void polygon3d_f(BITMAP *bmp, int type, BITMAP *texture, int vc, V3D_f *vtx[]);`

Draw 3d polygons onto the specified bitmap, using the specified rendering mode. Unlike the regular polygon() function, these routines don't support concave or self-intersecting shapes, and they can't draw onto mode-X screen bitmaps (if you want to write 3d code in mode-X, draw onto a memory bitmap and then blit to the screen). The width and height of the texture bitmap must be powers of two, but can be different, eg. a 64x16 texture is fine, but a 17x3 one is not. The vertex count parameter (vc) should be followed by an array containing the appropriate number of pointers to vertex structures: polygon3d() uses the fixed point V3D structure, while polygon3d_f() uses the floating point V3D_f structure. These are defined as:

```
typedef struct V3D
{
   fixed x, y, z;       - position
   fixed u, v;          - texture map coordinates
   int c;               - color
} V3D;
```

```
typedef struct V3D_f
{
   float x, y, z;       - position
   float u, v;          - texture map coordinates
   int c;               - color
} V3D_f;
```

How the vertex data is used depends on the rendering mode:

The x and y values specify the position of the vertex in 2d screen coordinates.

The z value is only required when doing perspective correct texture mapping, and specifies the depth of the point in 3d world coordinates.

The u and v coordinates are only required when doing texture mapping, and specify a point on the texture plane to be mapped on to this vertex. The texture plane is an infinite plane with the texture bitmap tiled across it. Each vertex in the polygon has a corresponding vertex on the texture plane, and the image of the resulting polygon in the texture plane will be mapped on to the polygon on the screen.

We refer to pixels in the texture plane as texels. Each texel is a block, not just a point, and whole numbers for u and v refer to the top-left corner of a texel. This has a few implications. If you want to draw a rectangular polygon and map a texture sized 32x32 on to it, you would use the texture coordinates (0,0), (0,32), (32,32) and (32,0), assuming the vertices are specified in anticlockwise order. The texture will then be mapped perfectly on to the polygon. However, note that when we set u=32, the last column of texels seen on the screen is the one at u=31, and the same goes for v. This is because the coordinates refer to the top-left corner of the texels. In effect, texture coordinates at the right and bottom on the texture plane are exclusive.

There is another interesting point here. If you have two polygons side by side sharing two vertices (like the two parts of folded piece of cardboard), and you want to map a texture across them seamlessly, the values of u and v on the vertices at the join will be the same for both polygons. For example, if they are both rectangular, one polygon may use (0,0), (0,32), (32,32) and (32,0), and the other may use (32,0), (32,32), (64,32), (64,0). This would create a seamless join.

Of course you can specify fractional numbers for u and v to indicate a point part-way across a texel. In addition, since the texture plane is infinite, you can specify larger values than the size of the texture. This can be used to tile the texture several times across the polygon.

The c value specifies the vertex color, and is interpreted differently by various rendering modes.

The type parameter specifies the polygon rendering mode, and can be any of the values:

POLYTYPE_FLAT:
A simple flat shaded polygon, taking the color from the c value of the first

vertex. This polygon type is affected by the drawing_mode() function, so it can be used to render XOR or translucent polygons.

POLYTYPE_GCOL:
A single-color gouraud shaded polygon. The colors for each vertex are taken from the c value, and interpolated across the polygon. This is very fast, but will only work in 256 color modes if your palette contains a smooth gradient between the colors. In truecolor modes it interprets the color as a packed, display-format value as produced by the makecol() function.

POLYTYPE_GRGB:
A gouraud shaded polygon which interpolates RGB triplets rather than a single color. In 256 color modes this uses the global rgb_map table to convert the result to an 8 bit paletted color, so it must only be used after you have set up the RGB mapping table! The colors for each vertex are taken from the c value, which is interpreted as a 24 bit RGB triplet (0xFF0000 is red, 0x00FF00 is green, and 0x0000FF is blue).

POLYTYPE_ATEX:
An affine texture mapped polygon. This stretches the texture across the polygon with a simple 2d linear interpolation, which is fast but not mathematically correct. It can look ok if the polygon is fairly small or flat-on to the camera, but because it doesn't deal with perspective foreshortening, it can produce strange warping artifacts. To see what I mean, run test.exe and see what happens to the polygon3d() test when you zoom in very close to the cube.

POLYTYPE_PTEX:
A perspective-correct texture mapped polygon. This uses the z value from the vertex structure as well as the u/v coordinates, so textures are displayed correctly regardless of the angle they are viewed from. Because it involves division calculations in the inner texture mapping loop, this mode is a lot slower than POLYTYPE_ATEX, and it uses floating point so it will be very slow on anything less than a Pentium (even with an FPU, a 486 can't overlap floating point division with other integer operations like the Pentium can).

POLYTYPE_ATEX_MASK:
POLYTYPE_PTEX_MASK:
Like POLYTYPE_ATEX and POLYTYPE_PTEX, but zero texture map pixels are skipped, allowing parts of the texture map to be transparent.

POLYTYPE_ATEX_LIT:
POLYTYPE_PTEX_LIT:
Like POLYTYPE_ATEX and POLYTYPE_PTEX, but the global color_map table (for 256 color modes) or blender function (for non-MMX truecolor modes) is used to blend the texture with a light level taken from the c value in the vertex structure. This must only be used after you have set up the color mapping table or blender functions!

POLYTYPE_ATEX_MASK_LIT:
POLYTYPE_PTEX_MASK_LIT:
Like POLYTYPE_ATEX_LIT and POLYTYPE_PTEX_LIT, but zero texture map pixels are skipped, allowing parts of the texture map to be transparent.

POLYTYPE_ATEX_TRANS:
POLYTYPE_PTEX_TRANS:
Render translucent textures. All the general rules for drawing translucent things apply. However, these modes have a major limitation: they only work with memory bitmaps or linear frame buffers (not with banked frame buffers). Don't even try, they do not check and your program will die horribly (or at least draw wrong things).

POLYTYPE_ATEX_MASK_TRANS:
POLYTYPE_PTEX_MASK_TRANS:
Like POLYTYPE_ATEX_TRANS and POLYTYPE_PTEX_TRANS, but zero texture map pixels are skipped.

If the CPU_MMX flag of the cpu_capabilities global variable is set, the GRGB and truecolor *LIT routines will be optimised using MMX instructions. If the CPU_3DNOW flag is set, the truecolor PTEX*LIT routines will take advantage of the 3DNow! CPU extensions.

Using MMX for *LIT routines has a side effect: normally (without MMX), these routines use the blender functions used also for other lighting functions, set with set_trans_blender() or set_blender_mode(). The MMX versions only use the RGB value passed to set_trans_blender() and do the linear interpolation themselves. Therefore a new set of blender functions passed to set_blender_mode() is ignored.

See also:

See Section 18.2 [triangle3d], page 130.

See Section 18.3 [quad3d], page 131.

See Section 13.10 [polygon], page 107.

See Section 18.5 [clip3d], page 131.

See Section 1.21 [cpu_capabilities], page 7.

18.2 triangle3d

```
void triangle3d(BITMAP *bmp, int type, BITMAP *tex, V3D *v1, *v2, *v3);
void triangle3d_f(BITMAP *bmp, int type, BITMAP *tex, V3D_f *v1, *v2, *v3);
```

Draw 3d triangles, using either fixed or floating point vertex structures. Unlike quad3d[_f], triangle3d[_f] functions are not wrappers of polygon3d[_f]. The triangle3d[_f] functions use their own routines taking into account the constantness of the gradients. Therefore triangle3d[_f](bmp, type, tex, v1, v2, v3) is faster than polygon3d[_f](bmp, type, tex, 3, v[]).

See also:

See Section 18.1 [polygon3d], page 127.

See Section 18.3 [quad3d], page 131.

See Section 13.9 [triangle], page 107.

18.3 quad3d

`void quad3d(BITMAP *bmp, int type, BITMAP *tex, V3D *v1, *v2, *v3, *v4);`

`void quad3d_f(BITMAP *bmp, int type, BITMAP *tex, V3D_f *v1, *v2, *v3, *v4);`

Draw 3d quads, using either fixed or floating point vertex structures. These are equivalent to calling polygon3d(bmp, type, tex, 4, v[]); or polygon3d_f(bmp, type, tex, 4, v[]);

See also:

See Section 18.1 [polygon3d], page 127.

See Section 18.2 [triangle3d], page 130.

18.4 clip3d_f

`int clip3d_f(int type, float min_z, float max_z, int vc, const V3D_f *vtx[], V3D_f *vout[], V3D_f *vtmp[], int out[]);`

Clips the polygon given in vtx. The number of vertices is vc, the result goes in vout, and vtmp and out are needed for internal purposes. The pointers in vtx, vout and vtmp must point to valid V3D_f structures. As additional vertices may appear in the process of clipping, so the size of vout, vtmp and out should be at least vc * (1.5 ^ n), where n is the number of clipping planes (5 or 6), and '^' denotes "to the power of". The frustum (viewing volume) is defined by -z<x<z, -z<y<z, 0<min_z<z<max_z. If max_z<=min_z, the z<max_z clipping is not done. As you can see, clipping is done in the camera space, with perspective in mind, so this routine should be called after you apply the camera matrix, but before the perspective projection. The routine will correctly interpolate u, v, and c in the vertex structure. However, no provision is made for high/truecolor GCOL.

See also:

See Section 18.1 [polygon3d], page 127.

See Section 18.5 [clip3d], page 131.

18.5 clip3d

`int clip3d(int type, fixed min_z, fixed max_z, int vc, const V3D *vtx[], V3D *vout[], V3D *vtmp[], int out[]);`

Fixed point version of clip3d_f(). This function should be used with caution, due to the limited precision of fixed point arithmetic and high chance of rounding errors: the floating point code is better for most situations.

See also:

See Section 18.1 [polygon3d], page 127.

See Section 18.4 [clip3d_f], page 131.

18.6 zbuffered rendering

A Z-buffer stores the depth of each pixel that is drawn on a viewport. When a 3D object is rendered, the depth of each of its pixels is compared against the value stored into the Z-buffer: if the pixel is closer it is drawn, otherwise it is skipped.

No polygon sorting is needed. However, backface culling should be done because it prevents many invisible polygons being compared against the Z-buffer. Z-buffered rendering is the only algorithm supported by Allegro that directly solves penetrating shapes (see example exzbuf.c, for instance). The price to pay is more complex (and slower) routines.

Z-buffered polygons are designed as an extension of the normal POLYTYPE_* rendering styles. Just OR the POLYTYPE with the value POLYTYPE_ZBUF, and the normal polygon3d(), polygon3d_f(), quad3d(), etc. functions will render z-buffered polygons.

Example:

```
    polygon3d(bmp, POLYTYPE_ATEX | POLYTYPE_ZBUF, tex, vc, vtx);
```

Of course, the z coordinates have to be valid regardless of rendering style.

A Z-buffered rendering procedure looks like a double-buffered rendering procedure. You should follow four steps: create a Z-buffer at the beginning of the program and make the library use it by calling set_zbuffer(). Then, for each frame, clear the Z-buffer and draw polygons with POLYTYPE_* | POLYTYPE_ZBUF and finally destroy the Z-buffer when leaving the program.

Notes on Z-buffered renderers:

- Unlike the normal POLYTYPE_FLAT renderers, the Z-buffered ones don't use the hline() routine. Therefore DRAW_MODE has no effect.
- The *LIT* routines work the traditional way - through the set of blender routines.
- All the Z-buffered routines are much slower than their normal counterparts (they all use the FPU to interpolate and test 1/z values).

18.7 create_zbuffer

`ZBUFFER *create_zbuffer(BITMAP *bmp);`

Creates a Z-buffer using the size of the BITMAP you are planning to draw on. Several Z-buffers can be defined but only one can be used at the same time, so you must call set_zbuffer() to make this Z-buffer active.

See also:

See Section 18.8 [create_sub_zbuffer], page 133.

See Section 18.9 [set_zbuffer], page 133.

See Section 18.10 [clear_zbuffer], page 133.

See Section 18.11 [destroy_zbuffer], page 134.

18.8 create_sub_zbuffer

`ZBUFFER *create_sub_zbuffer(ZBUFFER *parent, int x, int y, int width, int height);`

Creates a sub-z-buffer, ie. a z-buffer sharing drawing memory with a pre-existing z-buffer, but possibly with a different size. The same rules as for sub-bitmaps apply: the sub-z-buffer width and height can extend beyond the right and bottom edges of the parent (they will be clipped), but the origin point must lie within the parent region.

When drawing z-buffered to a bitmap, the top left corner of the bitmap is always mapped to the top left corner of the current z-buffer. So this function is primarily useful if you want to draw to a sub-bitmap and use the corresponding sub-area of the z-buffer. In other cases, eg. if you just want to draw to a sub-bitmap of screen (and not to other parts of screen), then you would usually want to create a normal z-buffer (not sub-z-buffer) the size of the visible screen. You don't need to first create a z-buffer the size of the virtual screen and then a sub-z-buffer of that.

See also:

See Section 18.7 [create_zbuffer], page 132.

See Section 9.4 [create_sub_bitmap], page 81.

See Section 18.11 [destroy_zbuffer], page 134.

18.9 set_zbuffer

`void set_zbuffer(ZBUFFER *zbuf);`

Makes the given Z-buffer be the active one. This should have been previously created with create_zbuffer().

See also:

See Section 18.7 [create_zbuffer], page 132.

See Section 18.10 [clear_zbuffer], page 133.

See Section 18.11 [destroy_zbuffer], page 134.

18.10 clear_zbuffer

`void clear_zbuffer(ZBUFFER *zbuf, float z);`

Writes z into the given Z-buffer (0 means far away). This function should be used to initialize the Z-buffer before each frame. Actually, low-level routines compare depth of the current pixel with 1/z: for example, if you want to clip polygons farther than 10, you must call clear_zbuffer(zbuf, 0.1);

See also:

See Section 18.7 [create_zbuffer], page 132.

See Section 18.9 [set_zbuffer], page 133.

See Section 18.11 [destroy_zbuffer], page 134.

18.11 destroy_zbuffer

```
void destroy_zbuffer(ZBUFFER *zbuf);
```

Destroys the Z-buffer when you are finished with it.

See also:
See Section 18.7 [create_zbuffer], page 132.
See Section 18.9 [set_zbuffer], page 133.
See Section 18.10 [clear_zbuffer], page 133.

18.12 scene rendering

Allegro provides two simple approaches to remove hidden surfaces:

- Z-buffering - (see above)
- Scan-line algorithms - along each scanline on your screen, you keep track of what polygons you are "in" and which is the nearest. This status changes only where the scanline crosses some polygon edge. So you have to juggle an edge list and a polygon list. And you have to sort the edges for each scanline (this can be countered by keeping the order of the previous scanline - it won't change much). The BIG advantage is that you write each pixel only once. If you have a lot of overlapping polygons you can get incredible speeds compared to any of the previous algorithms. This algorithm is covered by the *_scene routines.

The scene rendering has approximately the following steps:

- Initialize the scene (set the clip area, clear the bitmap, blit a background, etc.)
- Call clear_scene().
- Transform all your points to camera space.
- Clip polygons.
- Project with persp_project() or persp_project_f().
- "Draw" polygons with scene_polygon3d() and/or scene_polygon3d_f(). This doesn't do any actual drawing, only initializes tables.
- Render all the polygons defined previously to the bitmap with render_scene().
- Overlay some non-3D graphics.
- Show the bitmap (blit it to screen, flip the page, etc).

For each horizontal line in the viewport an x-sorted edge list is used to keep track of what polygons are "in" and which is the nearest. Vertical coherency is used - the edge list for a scanline is sorted starting from the previous one - it won't change much. The scene rendering routines use the same low-level asm routines as normal polygon3d().

Notes on scene rendering:

- Unlike polygon3d(), scene_polygon3d() requires valid z coordinates for all vertices, regardless of rendering style (unlike polygon3d(), which only uses z coordinate for *PTEX*).
- All polygons passed to scene_polygon3d() have to be persp_project()'ed.
- After render_scene() the mode is reset to SOLID.

Using a lot of *MASK* polygons drastically reduces performance, because when a MASKed polygon is the first in line of sight, the polygons underneath have to be drawn too. The same applies to FLAT polygons drawn with DRAW_MODE_TRANS.

Z-buffered rendering works also within the scene renderer. It may be helpful when you have a few intersecting polygons, but most of the polygons may be safely rendered by the normal scanline sorting algo. Same as before: just OR the POLYTYPE with POLYTYPE_ZBUF. Also, you have to clear the z-buffer at the start of the frame. Example:

```
clear_scene(buffer);
if (some_polys_are_zbuf) clear_zbuffer(0.);
while (polygons) {
   ...
   if (this_poly_is_zbuf) type |= POLYTYPE_ZBUF;
   scene_polygon3d(type, tex, vc, vtx);
}
render_scene();
```

18.13 create_scene

`int create_scene(int nedge, int npoly);`

Allocates memory for a scene, nedge and npoly are your estimates of how many edges and how many polygons you will render (you cannot get over the limit specified here). If you use same values in succesive calls, the space will be reused (no new malloc()).

The memory allocated is a little less than 150 * (nedge + npoly) bytes. Returns zero on success, or a negative number if allocations fail.

See also:
See Section 18.16 [scene_polygon3d], page 136.
See Section 18.17 [render_scene], page 137.
See Section 18.14 [clear_scene], page 135.
See Section 18.15 [destroy_scene], page 136.
See Section 18.18 [scene_gap], page 137.
See Section 18.7 [create_zbuffer], page 132.

18.14 clear_scene

`void clear_scene(BITMAP *bmp);`

Initializes a scene. The bitmap is the bitmap you will eventually render on.

See also:
See Section 18.13 [create_scene], page 135.
See Section 18.16 [scene_polygon3d], page 136.
See Section 18.17 [render_scene], page 137.
See Section 18.15 [destroy_scene], page 136.
See Section 18.18 [scene_gap], page 137.

18.15 destroy_scene

`void destroy_scene();`

Deallocate memory previously allocated by create_scene.

See also:
See Section 18.13 [create_scene], page 135.
See Section 18.16 [scene_polygon3d], page 136.
See Section 18.14 [clear_scene], page 135.
See Section 18.17 [render_scene], page 137.
See Section 18.18 [scene_gap], page 137.

18.16 scene_polygon3d

`int scene_polygon3d(int type, BITMAP *texture, int vc, V3D *vtx[]);`

`int scene_polygon3d_f(int type, BITMAP *texture, int vc, V3D_f *vtx[]);`

Puts a polygon in the rendering list. Nothing is really rendered at this moment. Should be called between clear_scene() and render_scene().

Arguments are the same as for polygon3d(), except the bitmap is missing. The one passed to clear_scene() will be used.

Unlike polygon3d(), the polygon may be concave or self-intersecting. Shapes that penetrate one another may look OK, but they are not really handled by this code.

Note that the texture is stored as a pointer only, and you should keep the actual bitmap around until render_scene(), where it is used.

Since the FLAT style is implemented with the low-level hline() funtion, the FLAT style is subject to DRAW_MODEs. All these modes are valid. Along with the polygon, this mode will be stored for the rendering moment, and also all the other related variables (color_map pointer, pattern pointer, anchor, blender values).

The settings of the CPU_MMX and CPU_3DNOW flags of the cpu_capabilities global variable on entry in this routine affect the choice of low-level asm routine that will be used by render_scene() for this polygon.

Returns zero on success, or a negative number if it won't be rendered for lack of a rendering routine.

See also:
See Section 18.13 [create_scene], page 135.
See Section 18.14 [clear_scene], page 135.
See Section 18.17 [render_scene], page 137.
See Section 18.15 [destroy_scene], page 136.
See Section 18.1 [polygon3d], page 127.
See Section 1.21 [cpu_capabilities], page 7.

18.17 render_scene

```
void render_scene();
```

Renders all the specified scene_polygon3d()'s on the bitmap passed to clear_scene(). Rendering is done one scanline at a time, with no pixel being processed more than once.

Note that between clear_scene() and render_scene() you shouldn't change the clip rectangle of the destination bitmap. For speed reasons, you should set the clip rectangle to the minimum.

Note also that all the textures passed to scene_polygon3d() are stored as pointers only and actually used in render_scene().

See also:
See Section 18.13 [create_scene], page 135.
See Section 18.14 [clear_scene], page 135.
See Section 18.15 [destroy_scene], page 136.
See Section 18.18 [scene_gap], page 137.
See Section 18.16 [scene_polygon3d], page 136.

18.18 scene_gap

```
extern float scene_gap;
```

This number (default value = 100.0) controls the behaviour of the z-sorting algorithm. When an edge is very close to another's polygon plane, there is an interval of uncertainty in which you cannot tell which object is visible (which z is smaller). This is due to cumulative numerical errors for edges that have undergone a lot of transformations and interpolations.

The default value means that if the 1/z values (in projected space) differ by only 1/100 (one percent), they are considered to be equal and the x-slopes of the planes are used to find out which plane is getting closer when we move to the right.

Larger values means narrower margins, and increasing the chance of missing true adjacent edges/planes. Smaller values means larger margins, and increasing the chance of mistaking close polygons for adjacent ones. The value of 100 is close to the optimum. However, the optimum shifts slightly with resolution, and may be application-dependent. It is here for you to fine-tune.

See also:
See Section 18.13 [create_scene], page 135.
See Section 18.14 [clear_scene], page 135.
See Section 18.15 [destroy_scene], page 136.
See Section 18.17 [render_scene], page 137.
See Section 18.16 [scene_polygon3d], page 136.

19 Transparency and patterned drawing

19.1 drawing_mode

`void drawing_mode(int mode, BITMAP *pattern, int x_anchor, int y_anchor);`

Sets the graphics drawing mode. This only affects the geometric routines like putpixel, lines, rectangles, circles, polygons, floodfill, etc, not the text output, blitting, or sprite drawing functions. The mode should be one of the following constants:

```
     DRAW_MODE_SOLID              - the default, solid color
                                    drawing
     DRAW_MODE_XOR                - exclusive-or drawing
     DRAW_MODE_COPY_PATTERN       - multicolored pattern fill
     DRAW_MODE_SOLID_PATTERN      - single color pattern fill
     DRAW_MODE_MASKED_PATTERN     - masked pattern fill
     DRAW_MODE_TRANS              - translucent color blending
```

In DRAW_MODE_SOLID, pixels of the bitmap being drawn onto are simply replaced by those produced by the drawing function.

In DRAW_MODE_XOR, pixels are written to the bitmap with an exclusive-or operation rather than a simple copy, so drawing the same shape twice will erase it. Because it involves reading as well as writing the bitmap memory, xor drawing is a lot slower than the normal replace mode.

With the patterned modes, you provide a pattern bitmap which is tiled across the surface of the shape. Allegro stores a pointer to this bitmap rather than copying it, so you must not destroy the bitmap while it is still selected as the pattern. The width and height of the pattern must be powers of two, but they can be different, eg. a 64x16 pattern is fine, but a 17x3 one is not. The pattern is tiled in a grid starting at point (x_anchor, y_anchor). Normally you should just pass zero for these values, which lets you draw several adjacent shapes and have the patterns meet up exactly along the shared edges. Zero alignment may look peculiar if you are moving a patterned shape around the screen, however, because the shape will move but the pattern alignment will not, so in some situations you may wish to alter the anchor position.

When you select DRAW_MODE_COPY_PATTERN, pixels are simply copied from the pattern bitmap onto the destination bitmap. This allows the use of multicolored patterns, and means that the color you pass to the drawing routine is ignored. This is the fastest of the patterned modes.

In DRAW_MODE_SOLID_PATTERN, each pixel in the pattern bitmap is compared with the mask color, which is zero in 256 color modes or bright pink for truecolor data (maximum red and blue, zero green). If the pattern pixel is solid, a pixel of the color you passed to the drawing routine is written to the destination bitmap, otherwise a zero is written. The pattern is thus treated as a monochrome bitmask, which lets you use the same pattern to draw different shapes in different colors, but prevents the use of multicolored patterns.

DRAW_MODE_MASKED_PATTERN is almost the same as DRAW_MODE_SOLID_PATTERN, but the masked pixels are skipped rather than being written as zeros, so the background shows through the gaps.

In DRAW_MODE_TRANS, the global color_map table or truecolor blender functions are used to overlay pixels on top of the existing image. This must only be used after you have set up the color mapping table (for 256 color modes) or blender functions (for truecolor modes). Because it involves reading as well as writing the bitmap memory, translucent drawing is very slow if you draw directly to video RAM, so wherever possible you should use a memory bitmap instead.

See also:
See Section 19.2 [xor_mode], page 139.
See Section 19.3 [solid_mode], page 139.
See Section 19.5 [color_map], page 140.
See Section 19.11 [set_trans_blender], page 143.

19.2 xor_mode

`void xor_mode(int on);`

This is a shortcut for toggling xor drawing mode on and off. Calling xor_mode(TRUE) is equivalent to drawing_mode (DRAW_MODE_XOR, NULL, 0, 0); Calling xor_mode(FALSE) is equivalent to drawing_mode(DRAW_MODE_SOLID, NULL, 0, 0);

See also:
See Section 19.1 [drawing_mode], page 138.

19.3 solid_mode

`void solid_mode();`

This is a shortcut for selecting solid drawing mode. It is equivalent to calling drawing_mode(DRAW_MODE_SOLID, NULL, 0, 0);

See also:

See Section 19.1 [drawing_mode], page 138.

19.4 256 color transparency

In paletted video modes, translucency and lighting are implemented with a 64k lookup table, which contains the result of combining any two colors c1 and c2. You must set up this table before you use any of the translucency or lighting routines. Depending on how you construct the table, a range of different effects are possible. For example, translucency can be implemented by using a color halfway between c1 and c2 as the result of the combination. Lighting is achieved by treating one of the colors as a light level (0-255) rather than a color, and setting up the table appropriately. A range of specialised effects are possible, for instance replacing any color with any other color and making individual source or destination colors completely solid or invisible.

Color mapping tables can be precalculated with the colormap utility, or generated at run-time. The COLOR_MAP structure is defined as:

```
typedef struct {
   unsigned char data[PAL_SIZE][PAL_SIZE];
} COLOR_MAP;
```

19.5 color_map

`extern COLOR_MAP *color_map;`

Global pointer to the color mapping table. This must be set before using any translucent or lit drawing functions in a 256 color video mode!

See also:

See Section 19.8 [create_color_table], page 141.

See Section 19.7 [create_light_table], page 141.

See Section 19.6 [create_trans_table], page 140.

See Section 19.9 [create_blender_table], page 142.

See Section 19.11 [set_trans_blender], page 143.

See Section 14.11 [draw_trans_sprite], page 116.

See Section 14.12 [draw_lit_sprite], page 116.

See Section 14.13 [draw_gouraud_sprite], page 117.

See Section 19.1 [drawing_mode], page 138.

19.6 create_trans_table

`void create_trans_table(COLOR_MAP *table, const PALETTE pal, int r, g, b, void (*callback)(int pos));`

Fills the specified color mapping table with lookup data for doing translucency effects with the specified palette. When combining the colors c1 and c2 with this table, the result will be a color somewhere between the two. The r, g, and b parameters specify the solidity of each color component, ranging from 0 (totally transparent) to 255 (totally solid). For 50% solidity, pass 128. This function treats source color #0 as a special case, leaving the destination unchanged whenever a zero source pixel is encountered, so that masked sprites will draw correctly. If the callback function is not NULL, it will be called 256 times during the calculation, allowing you to display a progress indicator.

See also:

See Section 19.5 [color_map], page 140.

See Section 19.7 [create_light_table], page 141.

See Section 19.8 [create_color_table], page 141.

See Section 19.9 [create_blender_table], page 142.

See Section 14.11 [draw_trans_sprite], page 116.

See Section 14.12 [draw_lit_sprite], page 116.

See Section 14.13 [draw_gouraud_sprite], page 117.

19.7 create_light_table

```
void create_light_table(COLOR_MAP *table, const PALETTE pal, int r, g, b,
void (*callback)(int pos));
```

Fills the specified color mapping table with lookup data for doing lighting effects with the specified palette. When combining the colors c1 and c2 with this table, c1 is treated as a light level from 0-255. At light level 255 the table will output color c2 unchanged, at light level 0 it will output the r, g, b value you specify to this function, and at intermediate light levels it will output a color somewhere between the two extremes. The r, g, and b values are in the range 0-63. If the callback function is not NULL, it will be called 256 times during the calculation, allowing you to display a progress indicator.

See also:

See Section 19.5 [color_map], page 140.

See Section 19.6 [create_trans_table], page 140.

See Section 19.8 [create_color_table], page 141.

See Section 19.9 [create_blender_table], page 142.

See Section 14.11 [draw_trans_sprite], page 116.

See Section 14.12 [draw_lit_sprite], page 116.

See Section 14.13 [draw_gouraud_sprite], page 117.

19.8 create_color_table

```
void create_color_table(COLOR_MAP *table, const PALETTE pal, void
(*blend)(PALETTE pal, int x, int y, RGB *rgb), void (*callback)(int pos));
```

Fills the specified color mapping table with lookup data for doing customised effects with the specified palette, calling the blend function to determine the results of each color combination. Your blend routine will be passed a pointer to the palette and the two colors which are to be combined, and should fill in the RGB structure with the desired result in 0-63 format. Allegro will then search the palette for the closest match to the RGB color that you requested, so it doesn't matter if the palette has no exact match for this color. If the callback function is not NULL, it will be called 256 times during the calculation, allowing you to display a progress indicator.

See also:

See Section 19.5 [color_map], page 140.

See Section 19.7 [create_light_table], page 141.

See Section 19.6 [create_trans_table], page 140.

See Section 19.9 [create_blender_table], page 142.

See Section 14.11 [draw_trans_sprite], page 116.

See Section 14.12 [draw_lit_sprite], page 116.

See Section 14.13 [draw_gouraud_sprite], page 117.

19.9 create_blender_table

```
void create_blender_table(COLOR_MAP *table, const PALETTE pal, void
(*callback)(int pos));
```

Fills the specified color mapping table with lookup data for doing a paletted equivalent of whatever truecolor blender mode is currently selected. After calling set_trans_blender(), set_blender_mode(), or any of the other truecolor blender mode routines, you can use this function to create an 8 bit mapping table that will have the same results as whatever 24 bit blending mode you have enabled.

See also:

See Section 19.5 [color_map], page 140.

See Section 19.7 [create_light_table], page 141.

See Section 19.6 [create_trans_table], page 140.

See Section 19.8 [create_color_table], page 141.

See Section 14.11 [draw_trans_sprite], page 116.

See Section 14.12 [draw_lit_sprite], page 116.

See Section 14.13 [draw_gouraud_sprite], page 117.

See Section 19.11 [set_trans_blender], page 143.

See Section 19.26 [set_blender_mode], page 148.

19.10 truecolor transparency

In truecolor video modes, translucency and lighting are implemented by a blender function of the form:

```
     unsigned long (*BLENDER_FUNC)(unsigned long x, y, n);
```

For each pixel to be drawn, this routine is passed two color parameters x and y, decomposes them into their red, green and blue components, combines them according to some mathematical transformation involving the interpolation factor n, and then merges the result back into a single return color value, which will be used to draw the pixel onto the destination bitmap.

The parameter x represents the blending modifier color and the parameter y represents the base color to be modified. The interpolation factor n is in the range [0-255] and controls the solidity of the blending.

When a translucent drawing function is used, x is the color of the source, y is the color of the bitmap begin drawn onto and n is the alpha level that was passed to the function that sets the blending mode (the RGB triplet that was passed to this function is not taken into account).

When a lit sprite drawing function is used, x is the color represented by the RGB triplet that was passed to the function that sets the blending mode (the alpha level that was passed to this function is not taken into account), y is the color of the sprite and n is the alpha level that was passed to the drawing function itself.

Since these routines may be used from various different color depths, there are three such callbacks, one for use with 15 bit 5.5.5 pixels, one for 16 bit 5.6.5 pixels, and one for 24 bit 8.8.8 pixels (this can be shared between the 24 and 32 bit code since the bit packing is the same).

19.11 set_trans_blender

```
void set_trans_blender(int r, int g, int b, int a);
```

Enables a linear interpolator blender mode for combining translucent or lit truecolor pixels.

See also:

See Section 19.26 [set_blender_mode], page 148.

See Section 19.12 [set_alpha_blender], page 144.

See Section 19.13 [set_write_alpha_blender], page 144.

See Section 19.5 [color_map], page 140.

See Section 14.11 [draw_trans_sprite], page 116.

See Section 14.12 [draw_lit_sprite], page 116.

See Section 19.1 [drawing_mode], page 138.

See Section 19.14 [set_add_blender], page 145.

See Section 19.15 [set_burn_blender], page 145.
See Section 19.16 [set_color_blender], page 145.
See Section 19.17 [set_difference_blender], page 145.
See Section 19.18 [set_dissolve_blender], page 146.
See Section 19.19 [set_dodge_blender], page 146.
See Section 19.20 [set_hue_blender], page 146.
See Section 19.21 [set_invert_blender], page 146.
See Section 19.22 [set_luminance_blender], page 147.
See Section 19.23 [set_multiply_blender], page 147.
See Section 19.24 [set_saturation_blender], page 147.
See Section 19.25 [set_screen_blender], page 147.

19.12 set_alpha_blender

`void set_alpha_blender();`

Enables the special alpha-channel blending mode, which is used for drawing 32 bit RGBA sprites. After calling this function, you can use draw_trans_sprite() or draw_trans_rle_sprite() to draw a 32 bit source image onto any hicolor or truecolor destination. The alpha values will be taken directly from the source graphic, so you can vary the solidity of each part of the image. You can't use any of the normal translucency functions while this mode is active, though, so you should reset to one of the normal blender modes (eg. set_trans_blender()) before drawing anything other than 32 bit RGBA sprites.

See also:
See Section 19.11 [set_trans_blender], page 143.
See Section 14.11 [draw_trans_sprite], page 116.
See Section 15.4 [draw_trans_rle_sprite], page 121.
See Section 19.13 [set_write_alpha_blender], page 144.

19.13 set_write_alpha_blender

`void set_write_alpha_blender();`

Enables the special alpha-channel editing mode, which is used for drawing alpha channels over the top of an existing 32 bit RGB sprite, to turn it into an RGBA format image. After calling this function, you can set the drawing mode to DRAW_MODE_TRANS and then write draw color values (0-255) onto a 32 bit image. This will leave the color values unchanged, but alter the alpha to whatever values you are writing. After enabling this mode you can also use draw_trans_sprite() to superimpose an 8 bit alpha mask over the top of an existing 32 bit sprite.

See also:

See Section 19.12 [set_alpha_blender], page 144.

See Section 14.11 [draw_trans_sprite], page 116.

See Section 19.1 [drawing_mode], page 138.

19.14 set_add_blender

```
void set_add_blender(int r, int g, int b, int a);
```

Enables an additive blender mode for combining translucent or lit truecolor pixels.

See also:

See Section 19.11 [set_trans_blender], page 143.

See Section 19.1 [drawing_mode], page 138.

19.15 set_burn_blender

```
void set_burn_blender(int r, int g, int b, int a);
```

Enables a burn blender mode for combining translucent or lit truecolor pixels. Here the lightness values of the colours of the source image reduce the lightness of the destination image, darkening the image.

See also:

See Section 19.11 [set_trans_blender], page 143.

See Section 19.1 [drawing_mode], page 138.

19.16 set_color_blender

```
void set_color_blender(int r, int g, int b, int a);
```

Enables a color blender mode for combining translucent or lit truecolor pixels. Applies only the hue and saturation of the source image to the destination image. The luminance of the destination image is not affected.

See also:

See Section 19.11 [set_trans_blender], page 143.

See Section 19.1 [drawing_mode], page 138.

19.17 set_difference_blender

```
void set_difference_blender(int r, int g, int b, int a);
```

Enables a difference blender mode for combining translucent or lit truecolor pixels. This makes an image which has colours calculated by the difference between the source and destination colours.

See also:

See Section 19.11 [set_trans_blender], page 143.

See Section 19.1 [drawing_mode], page 138.

19.18 set_dissolve_blender

`void set_dissolve_blender(int r, int g, int b, int a);`

Enables a dissolve blender mode for combining translucent or lit truecolor pixels. Randomly replaces the colours of some pixels in the destination image with those of the source image. The number of pixels replaced depends on the alpha value (higher value, more pixels replaced; you get the idea :).

See also:

See Section 19.11 [set_trans_blender], page 143.

See Section 19.1 [drawing_mode], page 138.

19.19 set_dodge_blender

`void set_dodge_blender(int r, int g, int b, int a);`

Enables a dodge blender mode for combining translucent or lit truecolor pixels. The lightness of colours in the source lighten the colours of the destination. White has the most effect; black has none.

See also:

See Section 19.11 [set_trans_blender], page 143.

See Section 19.1 [drawing_mode], page 138.

19.20 set_hue_blender

`void set_hue_blender(int r, int g, int b, int a);`

Enables a hue blender mode for combining translucent or lit truecolor pixels. This applies the hue of the source to the destination.

See also:

See Section 19.11 [set_trans_blender], page 143.

See Section 19.1 [drawing_mode], page 138.

19.21 set_invert_blender

`void set_invert_blender(int r, int g, int b, int a);`

Enables an invert blender mode for combining translucent or lit truecolor pixels. Blends the inverse (or negative) colour of the source with the destination.

See also:

See Section 19.11 [set_trans_blender], page 143.

See Section 19.1 [drawing_mode], page 138.

19.22 set_luminance_blender

`void set_luminance_blender(int r, int g, int b, int a);`

Enables a luminance blender mode for combining translucent or lit truecolor pixels. Applies the luminance of the source to the destination. The colour of the destination is not affected.

See also:

See Section 19.11 [set_trans_blender], page 143.

See Section 19.1 [drawing_mode], page 138.

19.23 set_multiply_blender

`void set_multiply_blender(int r, int g, int b, int a);`

Enables a multiply blender mode for combining translucent or lit truecolor pixels. Combines the source and destination images, multiplying the colours to produce a darker colour. If a colour is multiplied by white it remains unchanged; when multiplied by black it also becomes black.

See also:

See Section 19.11 [set_trans_blender], page 143.

See Section 19.1 [drawing_mode], page 138.

19.24 set_saturation_blender

`void set_saturation_blender(int r, int g, int b, int a);`

Enables a saturation blender mode for combining translucent or lit truecolor pixels. Applies the saturation of the source to the destination image.

See also:

See Section 19.11 [set_trans_blender], page 143.

See Section 19.1 [drawing_mode], page 138.

19.25 set_screen_blender

`void set_screen_blender(int r, int g, int b, int a);`

Enables a screen blender mode for combining translucent or lit truecolor pixels. This blender mode lightens the colour of the destination image by multiplying the inverse of the source and destination colours. Sort of like the opposite of the multiply blender mode.

See also:
See Section 19.11 [set_trans_blender], page 143.
See Section 19.1 [drawing_mode], page 138.

19.26 set_blender_mode

```
void set_blender_mode(BLENDER_FUNC b15, b16, b24, int r, g, b, a);
```

Specifies a custom set of truecolor blender routines, which can be used to implement whatever special interpolation modes you need. This function shares a single blender between the 24 and 32 bit modes.

See also:
See Section 19.27 [set_blender_mode_ex], page 148.
See Section 19.11 [set_trans_blender], page 143.
See Section 19.5 [color_map], page 140.
See Section 14.11 [draw_trans_sprite], page 116.
See Section 14.12 [draw_lit_sprite], page 116.
See Section 19.1 [drawing_mode], page 138.

19.27 set_blender_mode_ex

```
void set_blender_mode_ex(BLENDER_FUNC b15, b16, b24, b32, b15x, b16x, b24x,
int r, g, b, a);
```

Like set_blender_mode(), but allows you to specify a more complete set of blender routines. The b15, b16, b24, and b32 routines are used when drawing pixels onto destinations of the same format, while b15x, b16x, and b24x are used by draw_trans_sprite() and draw_trans_rle_sprite() when drawing RGBA images onto destination bitmaps of another format. These blenders will be passed a 32 bit x parameter, along with a y value of a different color depth, and must try to do something sensible in response.

See also:
See Section 19.26 [set_blender_mode], page 148.
See Section 19.12 [set_alpha_blender], page 144.

20 Converting between color formats

In general, Allegro is designed to be used in only one color depth at a time, so you will call set_color_depth() once and then store all your bitmaps in the same format. If you want to mix several different pixel formats, you can use create_bitmap_ex() in place of create_bitmap(), and call bitmap_color_depth() to query the format of a specific image. Most of the graphics routines require all their input parameters to be in the same format (eg. you cannot stretch a 15 bit source bitmap onto a 24 bit destination), but there are some exceptions: blit() and the rotation routines can copy between bitmaps of any format, converting the data as required, draw_sprite() can draw 256 color source images onto destinations of any format, draw_character() _always_ uses a 256 color source bitmap, whatever the format of the destination, the draw_trans_sprite() and draw_trans_rle_sprite() functions are able to draw 32 bit RGBA images onto any hicolor or truecolor destination, as long as you call set_alpha_blender() first, and the draw_trans_sprite() function is able to draw an 8 bit alpha channel image over the top of an existing 32 bit image, as long as you call set_write_alpha_blender() first.

Expanding a 256 color source onto a truecolor destination is fairly fast (obviously you must set the correct palette before doing this conversion!). Converting between different truecolor formats is slightly slower, and reducing truecolor images to a 256 color destination is very slow (it can be sped up significantly if you set up the global rgb_map table before doing the conversion).

20.1 bestfit_color

`int bestfit_color(const PALETTE pal, int r, int g, int b);`

Searches the specified palette for the closest match to the requested color, which are specified in the VGA hardware 0-63 format. Normally you should call makecol8() instead, but this lower level function may be useful if you need to use a palette other than the currently selected one, or specifically don't want to use the rgb_map lookup table.

See also:

See Section 12.1 [makecol8], page 99.

20.2 rgb_map

`extern RGB_MAP *rgb_map;`

To speed up reducing RGB values to 8 bit paletted colors, Allegro uses a 32k lookup table (5 bits for each color component). You must set up this table before using the gouraud shading routines, and if present the table will also vastly accelerate the makecol8() function. RGB tables can be precalculated with the rgbmap utility, or generated at runtime. The RGB_MAP structure is defined as:

```
typedef struct {
   unsigned char data[32][32][32];
} RGB_MAP;
```

See also:

See Section 20.3 [create_rgb_table], page 150.

See Section 12.1 [makecol8], page 99.

20.3 create_rgb_table

`void create_rgb_table(RGB_MAP *table, const PALETTE pal, void (*callback)(int pos));`

Fills the specified RGB mapping table with lookup data for the specified palette. If the callback function is not NULL, it will be called 256 times during the calculation, allowing you to display a progress indicator.

See also:
See Section 20.2 [rgb_map], page 149.

20.4 hsv_to_rgb

`void hsv_to_rgb(float h, float s, float v, int *r, int *g, int *b);`

`void rgb_to_hsv(int r, int g, int b, float *h, float *s, float *v);`

Convert color values between the HSV and RGB colorspaces. The RGB values range from 0 to 255, hue is from 0 to 360, and saturation and value are from 0 to 1.

21 Direct access to video memory

The bitmap structure looks like:

```
   typedef struct BITMAP
   {
      int w, h;               - size of the bitmap in pixels
      int clip;               - non-zero if clipping is turned on
      int cl, cr, ct, cb;     - clip rectangle left, right, top, and bottom
      int seg;                - segment for use with the line pointers
      unsigned char *line[];  - pointers to the start of each line
   } BITMAP;
```

There is some other stuff in the structure as well, but it is liable to change and you shouldn't use anything except the above. The clipping rectangle is inclusive on the left and top (0 allows drawing to position 0) but exclusive on the right and bottom (10 allows drawing to position 9, but not to 10). Note this is not the same format as you pass to set_clip(), which takes inclusive coordinates for all four corners.

There are several ways to get direct access to the image memory of a bitmap, varying in complexity depending on what sort of bitmap you are using.

The simplest approach will only work with memory bitmaps (obtained from create_bitmap(), grabber datafiles, and image files) and sub-bitmaps of memory bitmaps. This uses a table of char pointers, called 'line', which is a part of the bitmap structure and contains pointers to the start of each line of the image. For example, a simple memory bitmap putpixel function is:

```
     void memory_putpixel(BITMAP *bmp, int x, int y, int color)
     {
        bmp->line[y][x] = color;
     }
```

For truecolor modes you need to cast the line pointer to the appropriate type, for example:

```
     void memory_putpixel_15_or_16_bpp(BITMAP *bmp, int x, int y, int color)
     {
        ((short *)bmp->line[y])[x] = color;
     }

     void memory_putpixel_32(BITMAP *bmp, int x, int y, int color)
     {
        ((long *)bmp->line[y])[x] = color;
     }
```

If you want to write to the screen as well as to memory bitmaps, you need to use some helper macros, because the video memory may not be part of your normal address space. This simple routine will work for any linear screen, eg. a VESA linear framebuffers:

```
     void linear_screen_putpixel(BITMAP *bmp, int x, int y, int color)
     {
        bmp_select(bmp);
        bmp_write8((unsigned long)bmp->line[y]+x, color);
     }
```

For truecolor modes you should replace the bmp_write8() with bmp_write16(), bmp_write24(), or bmp_write32(), and multiply the x offset by the number of bytes per pixel. There are of course similar functions to read a pixel value from a bitmap, namely bmp_read8(), bmp_read16(), bmp_read24() and bmp_read32().

This still won't work in banked SVGA modes, however, or on platforms like Windows that do special processing inside the bank switching functions. For more flexible access to bitmap memory, you need to call the routines:

`unsigned long bmp_write_line(BITMAP *bmp, int line);`
: Selects the line of a bitmap that you are going to draw onto.

`unsigned long bmp_read_line(BITMAP *bmp, int line);`
: Selects the line of a bitmap that you are going to read from.

`unsigned long bmp_unwrite_line(BITMAP *bmp);`
: Releases the bitmap memory after you are finished with it. You only need to call this once at the end of a drawing operation, even if you have called bmp_write_line() or bmp_read_line() several times before it.

These are implemented as inline assembler routines, so they are not as inefficient as they might seem. If the bitmap doesn't require bank switching (ie. it is a memory bitmap, mode 13h screen, etc), these functions just return bmp->line[line].

Although SVGA bitmaps are banked, Allegro provides linear access to the memory within each scanline, so you only need to pass a y coordinate to these functions. Various x positions can be obtained by simply adding the x coordinate to the returned address. The return value is an unsigned long rather than a char pointer because the bitmap memory may not be in your data segment, and you need to access it with far pointers. For example, a putpixel using the bank switching functions is:

```
void banked_putpixel(BITMAP *bmp, int x, int y, int color)
{
   unsigned long address = bmp_write_line(bmp, y);
   bmp_select(bmp);
   bmp_write8(address+x, color);
   bmp_unwrite_line(bmp);
}
```

You will notice that Allegro provides separate functions for setting the read and write banks. It is important that you distinguish between these, because on some graphics cards the banks can be set individually, and on others the video memory is read and written at different addresses. Life is never quite as simple as we might wish it to be, though (this is true even when we _aren't_ talking about graphics coding :-) and so of course some cards only provide a single bank. On these the read and write bank functions will behave identically, so you shouldn't assume that you can read from one part of video memory and write to another at the same time. You can call bmp_read_line(), and read whatever you like from that line, and then call bmp_write_line() with the same or a different line number, and write whatever you like to this second line, but you mustn't call bmp_read_line() and bmp_write_line() together and expect to be able to read one line and write the other simultaneously. It would be nice if this was possible, but if you do it, your code won't work on single banked SVGA cards.

And then there's mode-X. If you've never done any mode-X graphics coding, you probably won't understand this, but for those of you who want to know how Allegro sets up the mode-X screen bitmaps, here goes...

The line pointers are still present, and they contain planar addresses, ie. the actual location at which you access the first pixel in the line. These addresses are guaranteed to be quad aligned, so you can just set the write plane, divide your x coordinate by four, and add it to the line pointer. For example, a mode-X putpixel is:

```
void modex_putpixel(BITMAP *b, int x, int y, int color)
{
   outportw(0x3C4, (0x100<<(x&3))|2);
   bmp_select(bmp);
   bmp_write8((unsigned long)bmp->line[y]+(x>>2), color);
}
```

Oh yeah: the djgpp nearptr hack. Personally I don't like this very much because it disables memory protection and isn't portable to other platforms, but a lot of people swear by it because it can give you direct access to the screen memory via a normal C pointer. Warning: this method will only work with the djgpp library, when using VGA 13h or a linear framebuffer modes!

In your setup code:

```
#include <sys/nearptr.h>

unsigned char *screenmemory;
unsigned long screen_base_addr;

__djgpp_nearptr_enable();

__dpmi_get_segment_base_address(screen->seg, &screen_base_addr);

screenmemory = (unsigned char *)(screen_base_addr +
                                 screen->line[0] -
                                 __djgpp_base_address);
```

Then:

```
void nearptr_putpixel(int x, int y, int color)
{
   screenmemory[x + y*VIRTUAL_W] = color;
}
```

22 FLIC routines

There are two high level functions for playing FLI/FLC animations: play_fli(), which reads the data directly from disk, and play_memory_fli(), which uses data that has already been loaded into RAM. Apart from the different sources of the data, these two functions behave identically. They draw the animation onto the specified bitmap, which should normally be the screen. Frames will be aligned with the top left corner of the bitmap: if you want to position them somewhere else you will need to create a sub-bitmap for the FLI player to draw onto. If loop is set the player will cycle when it reaches the end of the file, otherwise it will play through the animation once and then return. If the callback function is not NULL it will be called once for each frame, allowing you to perform background tasks of your own. This callback should normally return zero: if it returns non-zero the player will terminate (this is the only way to stop an animation that is playing in looped mode). The FLI player returns FLI_OK if it reached the end of the file, FLI_ERROR if something went wrong, and the value returned by the callback function if that was what stopped it. If you need to distinguish between different return values, your callback should return positive integers, since FLI_OK is zero and FLI_ERROR is negative. Note that the FLI player will only work when the timer module is installed, and that it will alter the palette according to whatever palette data is present in the animation file.

Occasionally you may need more detailed control over how an FLI is played, for example if you want to superimpose a text scroller on top of the animation, or to play it back at a different speed. You could do both of these with the lower level functions described below.

22.1 play_fli

```
int play_fli(const char *filename, BITMAP *bmp, int loop, int
(*callback)());
```

Plays an Autodesk Animator FLI or FLC animation file, reading the data from disk as it is required.

See also:

See Section 22.2 [play_memory_fli], page 154.

See Section 5.1 [install_timer], page 48.

See Section 22.11 [fli_frame], page 156.

22.2 play_memory_fli

```
int play_memory_fli(const void *fli_data, BITMAP *bmp, int loop, int
(*callback)());
```

Plays an Autodesk Animator FLI or FLC animation, reading the data from a copy of the file which is held in memory. You can obtain the fli_data pointer by mallocing a block of memory and reading an FLI file into it, or by importing an FLI into a grabber datafile. Playing animations from memory is obviously faster than cueing them directly from disk, and is particularly useful with short, looped FLI's. Animations can easily get very large, though, so in most cases you will probably be better just using play_fli().

See also:

See Section 22.1 [play_fli], page 154.

See Section 5.1 [install_timer], page 48.

See Section 22.11 [fli_frame], page 156.

22.3 open_fli

```
int open_fli(const char *filename);
int open_memory_fli(const void *fli_data);
```

Open FLI files ready for playing, reading the data from disk or memory respectively. Return FLI_OK on success. Information about the current FLI is held in global variables, so you can only have one animation open at a time.

See also:

See Section 22.4 [close_fli], page 155.

See Section 22.5 [next_fli_frame], page 155.

22.4 close_fli

```
void close_fli();
```

Closes an FLI file when you have finished reading from it.

See also:

See Section 22.3 [open_fli], page 154.

22.5 next_fli_frame

```
int next_fli_frame(int loop);
```

Reads the next frame of the current animation file. If loop is set the player will cycle when it reaches the end of the file, otherwise it will return FLI_EOF. Returns FLI_OK on success, FLI_ERROR or FLI_NOT_OPEN on error, and FLI_EOF on reaching the end of the file. The frame is read into the global variables fli_bitmap and fli_palette.

See also:

See Section 22.3 [open_fli], page 154.

See Section 22.6 [fli_bitmap], page 155.

See Section 22.7 [fli_palette], page 155.

See Section 22.12 [fli_timer], page 157.

See Section 22.11 [fli_frame], page 156.

22.6 fli_bitmap

```
extern BITMAP *fli_bitmap;
```

Contains the current frame of the FLI/FLC animation.

See also:

See Section 22.5 [next_fli_frame], page 155.

See Section 22.8 [fli_bmp_dirty_from], page 156.

See Section 22.7 [fli_palette], page 155.

22.7 fli_palette

```
extern PALETTE fli_palette;
```

Contains the current FLI palette.

See also:

See Section 22.5 [next_fli_frame], page 155.

See Section 22.9 [fli_pal_dirty_from], page 156.

See Section 22.6 [fli_bitmap], page 155.

22.8 fli_bmp_dirty_from

`extern int fli_bmp_dirty_from;`

`extern int fli_bmp_dirty_to;`

These variables are set by next_fli_frame() to indicate which part of the fli_bitmap has changed since the last call to reset_fli_variables(). If fli_bmp_dirty_from is greater than fli_bmp_dirty_to, the bitmap has not changed, otherwise lines fli_bmp_dirty_from to fli_bmp_dirty_to (inclusive) have altered. You can use these when copying the fli_bitmap onto the screen, to avoid moving data unnecessarily.

See also:

See Section 22.6 [fli_bitmap], page 155.

See Section 22.10 [reset_fli_variables], page 156.

22.9 fli_pal_dirty_from

`extern int fli_pal_dirty_from;`

`extern int fli_pal_dirty_to;`

These variables are set by next_fli_frame() to indicate which part of the fli_palette has changed since the last call to reset_fli_variables(). If fli_pal_dirty_from is greater than fli_pal_dirty_to, the palette has not changed, otherwise colors fli_pal_dirty_from to fli_pal_dirty_to (inclusive) have altered. You can use these when updating the hardware palette, to avoid unnecessary calls to set_palette().

See also:

See Section 22.7 [fli_palette], page 155.

See Section 22.10 [reset_fli_variables], page 156.

22.10 reset_fli_variables

`void reset_fli_variables();`

Once you have done whatever you are going to do with the fli_bitmap and fli_palette, call this function to reset the fli_bmp_dirty_* and fli_pal_dirty_* variables.

See also:

See Section 22.8 [fli_bmp_dirty_from], page 156.

See Section 22.9 [fli_pal_dirty_from], page 156.

22.11 fli_frame

`extern int fli_frame;`

Global variable containing the current frame number in the FLI file. This is useful for synchronising other events with the animation, for instance you could check it in a play_fli() callback function and use it to trigger a sample at a particular point.

See also:

See Section 22.1 [play_fli], page 154.

See Section 22.2 [play_memory_fli], page 154.

See Section 22.5 [next_fli_frame], page 155.

22.12 fli_timer

`extern volatile int fli_timer;`

Global variable for timing FLI playback. When you open an FLI file, a timer interrupt is installed which increments this variable every time a new frame should be displayed. Calling next_fli_frame() decrements it, so you can test it and know that it is time to display a new frame if it is greater than zero.

See also:
See Section 5.1 [install_timer], page 48.
See Section 22.5 [next_fli_frame], page 155.

23 Sound init routines

23.1 detect_digi_driver

`int detect_digi_driver(int driver_id);`

Detects whether the specified digital sound device is available. Returns the maximum number of voices that the driver can provide, or zero if the hardware is not present. This function must be called _before_ install_sound().

See also:

See Section 23.5 [install_sound], page 160.

See Section 23.3 [reserve_voices], page 158.

See Section 34.3 [DIGI_*/DOS], page 232.

See Section 35.2 [DIGI_*/Windows], page 237.

See Section 36.3 [DIGI_*/Unix], page 244.

See Section 37.2 [DIGI_*/BeOS], page 246.

See Section 38.2 [DIGI_*/QNX], page 247.

23.2 detect_midi_driver

`int detect_midi_driver(int driver_id);`

Detects whether the specified MIDI sound device is available. Returns the maximum number of voices that the driver can provide, or zero if the hardware is not present. There are two special-case return values that you should watch out for: if this function returns -1 it is a note-stealing driver (eg. DIGMID) that shares voices with the current digital sound driver, and if it returns 0xFFFF it is an external device like an MPU-401 where there is no way to determine how many voices are available. This function must be called _before_ install_sound().

See also:

See Section 23.5 [install_sound], page 160.

See Section 23.3 [reserve_voices], page 158.

See Section 34.4 [MIDI_*/DOS], page 233.

See Section 35.3 [MIDI_*/Windows], page 237.

See Section 36.4 [MIDI_*/Unix], page 244.

See Section 37.3 [MIDI_*/BeOS], page 246.

See Section 38.3 [MIDI_*/QNX], page 247.

23.3 reserve_voices

`void reserve_voices(int digi_voices, int midi_voices);`

Call this function to specify the number of voices that are to be used by the digital and MIDI sound drivers respectively. This must be done _before_ calling install_sound(). If you reserve too many voices, subsequent calls to install_sound() will fail. How many voices are available depends on the driver, and in some cases you will actually get more than you reserve (eg. the FM synth drivers will always provide 9 voices on an OPL2 and 18 on an OPL3, and the SB digital driver will round the number of voices up to the nearest power of two). Pass negative values to restore the default settings. You should be aware that the sound quality is usually inversely related to how many voices you use, so don't reserve any more than you really need.

See also:

See Section 23.4 [set_volume_per_voice], page 158.

See Section 23.5 [install_sound], page 160.

See Section 23.1 [detect_digi_driver], page 157.

See Section 23.2 [detect_midi_driver], page 158.

23.4 set_volume_per_voice

`void set_volume_per_voice(int scale);`

By default, when you reserve more voices for the digital sound driver, Allegro will reduce the volume of each voice to compensate. This is done to avoid too much distortion. The default volume per voice is such that, if you reserve n voices, you can play up to n/2 normalised samples with centre panning without risking distortion. The exception is when you have fewer than 8 voices, where the volume remains the same as for 8 voices.

If the resultant output is either too loud or too quiet, this function can be used to adjust the volume of each voice. You should first check that your speakers are at a reasonable volume, Allegro's global volume is at maximum (see set_volume() below), and any other mixers such as the Windows Volume Control are set reasonably.

Once you are sure that Allegro's output level is unsuitable for your application, use this function to adjust it. This must be done _before_ calling install_sound(). Note that this function is currently only relevant for drivers that use the Allegro mixer (which is most of them).

If you pass 0 to this function, each centred sample will play at the maximum volume possible without distortion, as will all samples played through a mono driver. Samples at the extreme left and right will distort if played at full volume. If you wish to play panned samples at full volume without distortion, you should pass 1 to this function. Note: this is different from the function's behaviour in WIPs 3.9.34, 3.9.35 and 3.9.36. If you used this function under one of these WIPs, you will have to increase your parameter by one to get the same volume.

Each time you increase the parameter by one, the volume of each voice will halve. For example, if you pass 4, you can play up to 16 centred samples at maximum volume without distortion.

Here are the default values, dependent on the number of voices:

```
   1-8 voices - set_volume_per_voice(2)
    16 voices - set_volume_per_voice(3)
    32 voices - set_volume_per_voice(4)
    64 voices - set_volume_per_voice(5)
```

Of course this function does not override the volume you specify with play_sample() or voice_set_volume(). It simply alters the overall output of the program. If you play samples at lower volumes, or if they are not normalised, then you can play more of them without distortion.

Warning: Allegro uses a clipping table to clip the waveform. The table is big enough to accommodate a total output of up to 4 times the maximum possible without distortion. If your output goes above this limit, the wave will 'wrap around' (peaks become troughs and vice versa), thus distorting much more. You should be careful that this does not happen.

It is recommended that you hard-code the parameter into your program, rather than offering it to the user. The user can alter the volume with the configuration file instead, or you can provide for this with set_volume().

To restore volume per voice to its default behaviour, pass -1.

See also:
See Section 23.3 [reserve_voices], page 158.
See Section 23.7 [set_volume], page 161.
See Section 23.5 [install_sound], page 160.
See Section 23.1 [detect_digi_driver], page 157.
See Section 23.2 [detect_midi_driver], page 158.

23.5 install_sound

```
int install_sound(int digi, int midi, const char *cfg_path);
```

Initialises the sound module. You should normally pass DIGI_AUTODETECT and MIDI_AUTODETECT as the driver parameters to this function, in which case Allegro will read hardware settings from the current configuration file. This allows the user to select different values with the setup utility: see the config section for details. Alternatively, see the platform specific documentation for a list of the available drivers. The cfg_path parameter is only present for compatibility with previous versions of Allegro, and has no effect on anything. Returns zero if the sound is successfully installed, and -1 on failure. If it fails it will store a description of the problem in allegro_error.

See also:
See Section 23.6 [remove_sound], page 161.
See Section 23.3 [reserve_voices], page 158.
See Section 23.1 [detect_digi_driver], page 157.
See Section 23.2 [detect_midi_driver], page 158.
See Section 23.7 [set_volume], page 161.
See Section 24.7 [play_sample], page 163.
See Section 24.10 [voice control], page 164.
See Section 25.4 [play_midi], page 172.
See Section 26.1 [play_audio_stream], page 176.
See Section 27.1 [install_sound_input], page 177.
See Section 1.6 [allegro_error], page 2.
See Section 3.23 [standard config variables], page 36.
See Section 34.3 [DIGI_*/DOS], page 232.
See Section 35.2 [DIGI_*/Windows], page 237.
See Section 36.3 [DIGI_*/Unix], page 244.
See Section 37.2 [DIGI_*/BeOS], page 246.
See Section 38.2 [DIGI_*/QNX], page 247.
See Section 34.4 [MIDI_*/DOS], page 233.
See Section 35.3 [MIDI_*/Windows], page 237.

See Section 36.4 [MIDI_*/Unix], page 244.

See Section 37.3 [MIDI_*/BeOS], page 246.

See Section 38.3 [MIDI_*/QNX], page 247.

23.6 remove_sound

`void remove_sound();`

Cleans up after you are finished with the sound routines. You don't normally need to call this, because allegro_exit() will do it for you.

See also:

See Section 23.5 [install_sound], page 160.

See Section 1.3 [allegro_exit], page 1.

23.7 set_volume

`void set_volume(int digi_volume, int midi_volume);`

Alters the global sound output volume. Specify volumes for both digital samples and MIDI playback, as integers from 0 to 255, or pass a negative value to leave one of the settings unchanged. If possible this routine will use a hardware mixer to control the volume, otherwise it will tell the sample and MIDI players to simulate a mixer in software.

See also:
See Section 23.5 [install_sound], page 160.

24 Digital sample routines

24.1 load_sample

`SAMPLE *load_sample(const char *filename);`

Loads a sample from a file, returning a pointer to it, or NULL on error. At present this function supports both mono and stereo WAV and mono VOC files, in 8 or 16 bit formats.

See also:

See Section 24.5 [destroy_sample], page 162.

See Section 24.3 [load_voc], page 162.

See Section 24.2 [load_wav], page 162.

See Section 24.7 [play_sample], page 163.

See Section 24.10 [voice control], page 164.

24.2 load_wav

`SAMPLE *load_wav(const char *filename);`

Loads a sample from a RIFF WAV file.

See also:

See Section 24.1 [load_sample], page 161.

24.3 load_voc

`SAMPLE *load_voc(const char *filename);`

Loads a sample from a Creative Labs VOC file.

See also:

See Section 24.1 [load_sample], page 161.

24.4 create_sample

`SAMPLE *create_sample(int bits, int stereo, int freq, int len);`

Constructs a new sample structure of the specified type. The data field points to a block of waveform data: see the structure definition in allegro/digi.h for details.

See also:

See Section 24.1 [load_sample], page 161.

See Section 24.5 [destroy_sample], page 162.

24.5 destroy_sample

`void destroy_sample(SAMPLE *spl);`

Destroys a sample structure when you are done with it. It is safe to call this even when the sample might be playing, because it checks and will kill it off if it is active.

See also:

See Section 24.1 [load_sample], page 161.

24.6 lock_sample

`void lock_sample(SAMPLE *spl);`

Under DOS, locks all the memory used by a sample. You don't normally need to call this function because load_sample() and create_sample() do it for you.

See also:

See Section 24.1 [load_sample], page 161.

See Section 24.4 [create_sample], page 162.

24.7 play_sample

`int play_sample(const SAMPLE *spl, int vol, int pan, int freq, int loop);`

Triggers a sample at the specified volume, pan position, and frequency. The volume and pan range from 0 (min/left) to 255 (max/right). Frequency is relative rather than absolute: 1000 represents the frequency that the sample was recorded at, 2000 is twice this, etc. If the loop flag is set, the sample will repeat until you call stop_sample(), and can be manipulated while it is playing by calling adjust_sample(). Returns the voice number that was allocated for the sample (a non-negative number if successful).

See also:

See Section 23.5 [install_sound], page 160.

See Section 24.1 [load_sample], page 161.

See Section 24.8 [adjust_sample], page 163.

See Section 24.9 [stop_sample], page 163.

See Section 24.10 [voice control], page 164.

24.8 adjust_sample

`void adjust_sample(const SAMPLE *spl, int vol, int pan, int freq, int loop);`

Alters the parameters of a sample while it is playing (useful for manipulating looped sounds). You can alter the volume, pan, and frequency, and can also clear the loop flag, which will stop the sample when it next reaches the end of its loop. If there are several copies of the same sample playing, this will adjust the first one it comes across. If the sample is not playing it has no effect.

See also:

See Section 24.7 [play_sample], page 163.

24.9 stop_sample

`void stop_sample(const SAMPLE *spl);`

Kills off a sample, which is required if you have set a sample going in looped mode. If there are several copies of the sample playing, it will stop them all.

See also:

See Section 24.7 [play_sample], page 163.

24.10 voice control

If you need more detailed control over how samples are played, you can use the lower level voice functions rather than just calling play_sample(). This is rather more work, because you have to explicitly allocate and free the voices rather than them being automatically released when they finish playing, but allows far more precise specification of exactly how you want everything to sound. You may also want to modify a couple of fields from the sample structure:

```
      int priority;
         Ranging 0-255 (default 128), this controls how voices are
         allocated if you attempt to play more than the driver can handle.
         This may be used to ensure that the less important sounds are
         cut off while the important ones are preserved.

      unsigned long loop_start;
      unsigned long loop_end;
         Loop position in sample units, by default set to the start and end of
         the sample.
```

See also:

See Section 23.5 [install_sound], page 160.

See Section 24.11 [allocate_voice], page 164.

See Section 24.12 [deallocate_voice], page 165.

See Section 24.13 [reallocate_voice], page 165.

See Section 24.14 [release_voice], page 165.

See Section 24.15 [voice_start], page 166.

See Section 24.17 [voice_set_priority], page 166.

See Section 24.18 [voice_check], page 166.

See Section 24.20 [voice_set_position], page 167.

See Section 24.21 [voice_set_playmode], page 167.

See Section 24.23 [voice_set_volume], page 168.

See Section 24.27 [voice_set_frequency], page 169.

See Section 24.31 [voice_set_pan], page 170.

24.11 allocate_voice

`int allocate_voice(const SAMPLE *spl);`

Allocates a soundcard voice and prepares it for playing the specified sample, setting up sensible default parameters (maximum volume, centre pan, no change of pitch, no looping). When you are finished with the voice you must free it by calling deallocate_voice() or release_voice(). Returns the voice number, or -1 if no voices are available.

See also:
See Section 24.10 [voice control], page 164.
See Section 24.12 [deallocate_voice], page 165.
See Section 24.13 [reallocate_voice], page 165.
See Section 24.14 [release_voice], page 165.
See Section 24.1 [load_sample], page 161.

24.12 deallocate_voice

`void deallocate_voice(int voice);`

Frees a soundcard voice, stopping it from playing and releasing whatever resources it is using.

See also:
See Section 24.11 [allocate_voice], page 164.
See Section 24.16 [voice_stop], page 166.

24.13 reallocate_voice

`void reallocate_voice(int voice, const SAMPLE *spl);`

Switches an already-allocated voice to use a different sample. Calling reallocate_voice(voice, sample) is equivalent to:

```
deallocate_voice(voice);
voice = allocate_voice(sample);
```

See also:
See Section 24.11 [allocate_voice], page 164.
See Section 24.12 [deallocate_voice], page 165.
See Section 24.1 [load_sample], page 161.

24.14 release_voice

`void release_voice(int voice);`

Releases a soundcard voice, indicating that you are no longer interested in manipulating it. The sound will continue to play, and any resources that it is using will automatically be freed when it finishes. This is essentially the same as deallocate_voice(), but it waits for the sound to stop playing before taking effect.

See also:
See Section 24.11 [allocate_voice], page 164.

See Section 24.12 [deallocate_voice], page 165.

24.15 voice_start

`void voice_start(int voice);`

Activates a voice, using whatever parameters have been set for it.

See also:

See Section 24.10 [voice control], page 164.

See Section 24.11 [allocate_voice], page 164.

See Section 24.16 [voice_stop], page 166.

See Section 24.14 [release_voice], page 165.

24.16 voice_stop

`void voice_stop(int voice);`

Stops a voice, storing the current position and state so that it may later be resumed by calling voice_start().

See also:

See Section 24.15 [voice_start], page 166.

See Section 24.12 [deallocate_voice], page 165.

See Section 24.14 [release_voice], page 165.

24.17 voice_set_priority

`void voice_set_priority(int voice, int priority);`

Sets the priority of a voice (range 0-255). This is used to decide which voices should be chopped off, if you attempt to play more than the soundcard driver can handle.

See also:

See Section 24.10 [voice control], page 164.

24.18 voice_check

`SAMPLE *voice_check(int voice);`

Checks whether a voice is currently allocated. It returns a copy of the sample that the voice is using, or NULL if the voice is inactive (ie. it has been deallocated, or the release_voice() function has been called and the sample has then finished playing).

See also:

See Section 24.11 [allocate_voice], page 164.

See Section 24.15 [voice_start], page 166.

See Section 24.19 [voice_get_position], page 167.

24.19 voice_get_position

`int voice_get_position(int voice);`

Returns the current position of a voice, in sample units, or -1 if it has finished playing.

See also:

See Section 24.10 [voice control], page 164.

See Section 24.20 [voice_set_position], page 167.

24.20 voice_set_position

`void voice_set_position(int voice, int position);`

Sets the position of a voice, in sample units.

See also:

See Section 24.10 [voice control], page 164.

See Section 24.19 [voice_get_position], page 167.

See Section 24.21 [voice_set_playmode], page 167.

24.21 voice_set_playmode

`void voice_set_playmode(int voice, int playmode);`

Adjusts the loop status of the specified voice. This can be done while the voice is playing, so you can start a sample in looped mode (having set the loop start and end positions to the appropriate values), and then clear the loop flag when you want to end the sound, which will cause it to continue past the loop end, play the subsequent part of the sample, and finish in the normal way. The mode parameter is a bitfield containing the following values:

- PLAYMODE_PLAY
  Plays the sample a single time. This is the default if you don't set the loop flag.
- PLAYMODE_LOOP
  Loops repeatedly through the sample, jumping back to the loop start position upon reaching the loop end.
- PLAYMODE_FORWARD
  Plays the sample from beginning to end. This is the default if you don't set the backward flag.
- PLAYMODE_BACKWARD
  Reverses the direction of the sample. If you combine this with the loop

flag, the sample jumps to the loop end position upon reaching the loop start (ie. you do not need to reverse the loop start and end values when you play the sample in reverse).

- PLAYMODE_BIDIR
  When used in combination with the loop flag, causes the sample to change direction each time it reaches one of the loop points, so it alternates between playing forwards and in reverse.

See also:

See Section 24.10 [voice control], page 164.

24.22 voice_get_volume

```
int voice_get_volume(int voice);
```

Returns the current volume of the voice, range 0-255.

See also:

See Section 24.10 [voice control], page 164.

See Section 24.23 [voice_set_volume], page 168.

24.23 voice_set_volume

```
void voice_set_volume(int voice, int volume);
```

Sets the volume of the voice, range 0-255.

See also:

See Section 24.10 [voice control], page 164.

See Section 24.22 [voice_get_volume], page 168.

See Section 24.24 [voice_ramp_volume], page 168.

24.24 voice_ramp_volume

```
void voice_ramp_volume(int voice, int time, int endvol);
```

Starts a volume ramp (crescendo or diminuendo) from the current volume to the specified ending volume, lasting for time milliseconds.

See also:

See Section 24.10 [voice control], page 164.

See Section 24.23 [voice_set_volume], page 168.

24.25 voice_stop_volumeramp

```
void voice_stop_volumeramp(int voice);
```

Interrupts a volume ramp operation.

See also:

See Section 24.24 [voice_ramp_volume], page 168.

24.26 voice_get_frequency

`int voice_get_frequency(int voice);`

Returns the current pitch of the voice, in Hz.

See also:

See Section 24.10 [voice control], page 164.

See Section 24.27 [voice_set_frequency], page 169.

24.27 voice_set_frequency

`void voice_set_frequency(int voice, int frequency);`

Sets the pitch of the voice, in Hz.

See also:

See Section 24.10 [voice control], page 164.

See Section 24.26 [voice_get_frequency], page 169.

See Section 24.28 [voice_sweep_frequency], page 169.

24.28 voice_sweep_frequency

`void voice_sweep_frequency(int voice, int time, int endfreq);`

Starts a frequency sweep (glissando) from the current pitch to the specified ending pitch, lasting for time milliseconds.

See also:

See Section 24.10 [voice control], page 164.

See Section 24.27 [voice_set_frequency], page 169.

24.29 voice_stop_frequency_sweep

`void voice_stop_frequency_sweep(int voice);`

Interrupts a frequency sweep operation.

See also:

See Section 24.28 [voice_sweep_frequency], page 169.

24.30 voice_get_pan

`int voice_get_pan(int voice);`

Returns the current pan position, from 0 (left) to 255 (right).

See also:

See Section 24.10 [voice control], page 164.

See Section 24.31 [voice_set_pan], page 170.

24.31 voice_set_pan

`void voice_set_pan(int voice, int pan);`

Sets the pan position, ranging from 0 (left) to 255 (right).

See also:

See Section 24.10 [voice control], page 164.

See Section 24.30 [voice_get_pan], page 170.

See Section 24.32 [voice_sweep_pan], page 170.

24.32 voice_sweep_pan

`void voice_sweep_pan(int voice, int time, int endpan);`

Starts a pan sweep (left right movement) from the current position to the specified ending position, lasting for time milliseconds.

See also:

See Section 24.10 [voice control], page 164.

See Section 24.31 [voice_set_pan], page 170.

24.33 voice_stop_pan_sweep

`void voice_stop_pan_sweep(int voice);`

Interrupts a pan sweep operation.

See also:

See Section 24.32 [voice_sweep_pan], page 170.

24.34 voice_set_echo

`void voice_set_echo(int voice, int strength, int delay);`

Sets the echo parameters for a voice (not currently implemented).

See also:

See Section 24.10 [voice control], page 164.

24.35 voice_set_tremolo

```
void voice_set_tremolo(int voice, int rate, int depth);
```

Sets the tremolo parameters for a voice (not currently implemented).

See also:

See Section 24.10 [voice control], page 164.

24.36 voice_set_vibrato

```
void voice_set_vibrato(int voice, int rate, int depth);
```

Sets the vibrato parameters for a voice (not currently implemented).

See also:

See Section 24.10 [voice control], page 164.

25 MIDI music routines

25.1 load_midi

```
MIDI *load_midi(const char *filename);
```

Loads a MIDI file (handles both format 0 and format 1), returning a pointer to a MIDI structure, or NULL on error.

See also:

See Section 25.2 [destroy_midi], page 171.

See Section 25.4 [play_midi], page 172.

25.2 destroy_midi

```
void destroy_midi(MIDI *midi);
```

Destroys a MIDI structure when you are done with it. It is safe to call this even when the MIDI file might be playing, because it checks and will kill it off if it is active.

See also:

See Section 25.1 [load_midi], page 171.

25.3 lock_midi

```
void lock_midi(MIDI *midi);
```

Under DOS, locks all the memory used by a MIDI file. You don't normally need to call this function because load_midi() does it for you.

See also:

See Section 25.1 [load_midi], page 171.

25.4 play_midi

`int play_midi(MIDI *midi, int loop);`

Starts playing the specified MIDI file, first stopping whatever music was previously playing. If the loop flag is set, the data will be repeated until replaced with something else, otherwise it will stop at the end of the file. Passing a NULL pointer will stop whatever music is currently playing. Returns non-zero if an error occurs (this may happen if a patch-caching wavetable driver is unable to load the required samples, or at least it might in the future when somebody writes some patch-caching wavetable drivers :-)

See also:

See Section 23.5 [install_sound], page 160.

See Section 25.1 [load_midi], page 171.

See Section 25.5 [play_looped_midi], page 172.

See Section 25.6 [stop_midi], page 172.

See Section 25.7 [midi_pause], page 173.

See Section 25.9 [midi_seek], page 173.

See Section 25.12 [midi_pos], page 174.

See Section 25.14 [midi_msg_callback], page 175.

25.5 play_looped_midi

`int play_looped_midi(MIDI *midi, int loop_start, int loop_end);`

Starts playing a MIDI file with a user-defined loop position. When the player reaches the loop end position or the end of the file (loop_end may be -1 to only loop at EOF), it will wind back to the loop start point. Both positions are specified in the same beat number format as the midi_pos variable.

See also:

See Section 25.4 [play_midi], page 172.

See Section 25.13 [midi_loop_start], page 174.

25.6 stop_midi

`void stop_midi();`

Stops whatever music is currently playing. This is the same thing as calling play_midi(NULL, FALSE).

See also:
See Section 25.4 [play_midi], page 172.
See Section 25.7 [midi_pause], page 173.

25.7 midi_pause

```
void midi_pause();
```

Pauses the MIDI player.

See also:
See Section 25.4 [play_midi], page 172.
See Section 25.6 [stop_midi], page 172.
See Section 25.8 [midi_resume], page 173.
See Section 25.9 [midi_seek], page 173.

25.8 midi_resume

```
void midi_resume();
```

Resumes playback of a paused MIDI file.

See also:
See Section 25.7 [midi_pause], page 173.

25.9 midi_seek

```
int midi_seek(int target);
```

Seeks to the given midi_pos in the current MIDI file. If the target is earlier in the file than the current midi_pos it seeks from the beginning; otherwise it seeks from the current position. Returns zero if it could successfully seek to the requested position. Otherwise, a return value of 1 means it stopped playing, and midi_pos is set to the negative length of the MIDI file (so you can use this function to determine the length of a MIDI file). A return value of 2 means the MIDI file looped back to the start.

See also:
See Section 25.4 [play_midi], page 172.
See Section 25.12 [midi_pos], page 174.

25.10 midi_out

`void midi_out(unsigned char *data, int length);`

Streams a block of MIDI commands into the player in realtime, allowing you to trigger notes, jingles, etc, over the top of whatever MIDI file is currently playing.

See also:

See Section 23.5 [install_sound], page 160.

See Section 25.11 [load_midi_patches], page 174.

See Section 27.12 [midi_recorder], page 181.

25.11 load_midi_patches

`int load_midi_patches();`

Forces the MIDI driver to load the entire set of patches ready for use. You will not normally need to call this, because Allegro automatically loads whatever data is required for the current MIDI file, but you must call it before sending any program change messages via the midi_out() command. Returns non-zero if an error occurred.

See also:

See Section 23.5 [install_sound], page 160.

See Section 25.10 [midi_out], page 173.

25.12 midi_pos

`extern volatile long midi_pos;`

Stores the current position (beat number) in the MIDI file, or contains a negative number if no music is currently playing. Useful for synchronising animations with the music, and for checking whether a MIDI file has finished playing.

See also:

See Section 25.4 [play_midi], page 172.

See Section 25.14 [midi_msg_callback], page 175.

25.13 midi_loop_start

`extern long midi_loop_start;`

`extern long midi_loop_end;`

The loop start and end points, set by the play_looped_midi() function. These may safely be altered while the music is playing, but you should be sure they are always set to sensible values (start < end). If you are changing them both at the same time, make sure to alter them in the right order in case a MIDI interrupt happens to occur in between your two writes! Setting these values to -1 represents the start and end of the file respectively.

See also:
See Section 25.5 [play_looped_midi], page 172.

25.14 midi_msg_callback

`extern void (*midi_msg_callback)(int msg, int byte1, int byte2);`

`extern void (*midi_meta_callback)(int type, const unsigned char *data, int length);`

`extern void (*midi_sysex_callback)(const unsigned char *data, int length);`

Hook functions allowing you to intercept MIDI player events. If set to anything other than NULL, these routines will be called for each MIDI message, meta-event, and system exclusive data block respectively. They will execute in an interrupt handler context, so all the code and data they use should be locked, and they must not call any operating system functions. In general you just use these routines to set some flags and respond to them later in your mainline code.

See also:
See Section 25.4 [play_midi], page 172.

25.15 load_ibk

`int load_ibk(char *filename, int drums);`

Reads in a .IBK patch definition file for use by the Adlib driver. If drums is set, it will load it as a percussion patch set, otherwise it will use it as a replacement set of General MIDI instruments. You may call this before or after initialising the sound code, or can simply set the ibk_file and ibk_drum_file variables in the configuration file to have the data loaded automatically. Note that this function has no effect on any drivers other than the Adlib one! Returns non-zero on error.

See also:
See Section 23.5 [install_sound], page 160.

26 Audio stream routines

The audio stream functions are for playing digital sounds that are too big to fit in a regular SAMPLE structure, either because they are huge files that you want to load in pieces as the data is required, or because you are doing something clever like generating the waveform on the fly.

26.1 play_audio_stream

`AUDIOSTREAM *play_audio_stream(int len, bits, stereo, freq, vol, pan);`

This function creates a new audio stream and starts it playing. The length is the size of each transfer buffer (in samples), which should normally (but doesn't have to) be a power of two somewhere around 1k in size. Larger buffers are more efficient and require fewer updates, but result in more latency between you providing the data and it actually being played. The bits parameter must be 8 or 16, freq is the sample rate of the data in Hertz. The vol and pan values use the same 0-255 ranges as the regular sample playing functions. The stereo parameter should be set to 1 for stereo streams, or 0 otherwise. If you want to adjust the pitch, volume, or panning of a stream once it is playing, you can use the regular voice_*() functions with stream->voice as a parameter. The sample data is always in unsigned format, with stereo waveforms consisting of alternating left/right samples, left sample first.

See also:

See Section 23.5 [install_sound], page 160.

See Section 26.3 [get_audio_stream_buffer], page 176.

See Section 26.2 [stop_audio_stream], page 176.

26.2 stop_audio_stream

`void stop_audio_stream(AUDIOSTREAM *stream);`

Destroys an audio stream when it is no longer required.

See also:

See Section 26.1 [play_audio_stream], page 176.

26.3 get_audio_stream_buffer

`void *get_audio_stream_buffer(AUDIOSTREAM *stream);`

You must call this function at regular intervals while an audio stream is playing, to provide the next buffer of sample data (the smaller the stream buffer size, the more often it must be called). If it returns NULL, the stream is still playing the previous lot of data, so you don't need to do anything. If it returns a value, that is the location of the next buffer to be played, and you should load the appropriate number of samples (however many you specified when creating the stream) to that address, for example using an fread() from a disk file. After filling the buffer with data, call free_audio_stream_buffer() to indicate that the new data is now valid. Note that this function should not be called from a timer handler...

See also:

See Section 26.1 [play_audio_stream], page 176.

See Section 26.4 [free_audio_stream_buffer], page 177.

26.4 free_audio_stream_buffer

`void free_audio_stream_buffer(AUDIOSTREAM *stream);`

Call this function after get_audio_stream_buffer() returns a non-NULL address, to indicate that you have loaded a new block of samples to that location and the data is now ready to be played.

See also:
See Section 26.3 [get_audio_stream_buffer], page 176.

27 Recording routines

27.1 install_sound_input

`int install_sound_input(int digi, int midi);`

Initialises the sound recorder module, returning zero on success. You must install the normal sound playback system before calling this routine. The two card parameters should use the same constants as install_sound(), including DIGI_NONE and MIDI_NONE to disable parts of the module, or DIGI_AUTODETECT and MIDI_AUTODETECT to guess the hardware.

See also:
See Section 23.5 [install_sound], page 160.
See Section 27.8 [start_sound_input], page 179.
See Section 27.12 [midi_recorder], page 181.
See Section 3.23 [standard config variables], page 36.
See Section 34.3 [DIGI_*/DOS], page 232.
See Section 35.2 [DIGI_*/Windows], page 237.
See Section 36.3 [DIGI_*/Unix], page 244.
See Section 37.2 [DIGI_*/BeOS], page 246.
See Section 38.2 [DIGI_*/QNX], page 247.
See Section 34.4 [MIDI_*/DOS], page 233.
See Section 35.3 [MIDI_*/Windows], page 237.
See Section 36.4 [MIDI_*/Unix], page 244.
See Section 37.3 [MIDI_*/BeOS], page 246.
See Section 38.3 [MIDI_*/QNX], page 247.

27.2 remove_sound_input

`void remove_sound_input();`

Cleans up after you are finished with the sound input routines. You don't normally need to call this, because remove_sound() and/or allegro_exit() will do it for you.

See also:

See Section 27.1 [install_sound_input], page 177.

See Section 23.6 [remove_sound], page 161.

See Section 1.3 [allegro_exit], page 1.

27.3 get_sound_input_cap_bits

`int get_sound_input_cap_bits();`

Checks which sample formats are supported by the current audio input driver, returning one of the bitfield values:

0 = audio input not supported
8 = eight bit audio input is supported
16 = sixteen bit audio input is supported
24 = both eight and sixteen bit audio input are supported

See also:

See Section 27.8 [start_sound_input], page 179.

See Section 27.6 [get_sound_input_cap_parm], page 179.

See Section 27.5 [get_sound_input_cap_rate], page 178.

See Section 27.4 [get_sound_input_cap_stereo], page 178.

27.4 get_sound_input_cap_stereo

`int get_sound_input_cap_stereo();`

Checks whether the current audio input driver is capable of stereo recording.

See also:

See Section 27.8 [start_sound_input], page 179.

See Section 27.6 [get_sound_input_cap_parm], page 179.

See Section 27.3 [get_sound_input_cap_bits], page 178.

See Section 27.5 [get_sound_input_cap_rate], page 178.

27.5 get_sound_input_cap_rate

`int get_sound_input_cap_rate(int bits, int stereo);`

Returns the maximum possible sample frequency for recording in the specified format, or zero if these settings are not supported.

See also:

See Section 27.8 [start_sound_input], page 179.

See Section 27.6 [get_sound_input_cap_parm], page 179.

See Section 27.3 [get_sound_input_cap_bits], page 178.

See Section 27.4 [get_sound_input_cap_stereo], page 178.

27.6 get_sound_input_cap_parm

`int get_sound_input_cap_parm(int rate, int bits, int stereo);`

Checks whether the specified recording frequency, number of bits, and mono/stereo mode are supported by the current audio driver, returning one of the values:

0 = it is impossible to record in this format
1 = recording is possible, but audio output will be suspended
2 = recording is possible at the same time as playing other sounds
-n = sampling rate not supported, but rate 'n' would work instead

See also:

See Section 27.8 [start_sound_input], page 179.

See Section 27.3 [get_sound_input_cap_bits], page 178.

See Section 27.5 [get_sound_input_cap_rate], page 178.

See Section 27.4 [get_sound_input_cap_stereo], page 178.

27.7 set_sound_input_source

`int set_sound_input_source(int source);`

Selects the audio input source, returning zero on success or -1 if the hardware does not provide an input select register. The parameter should be one of the values:

SOUND_INPUT_MIC
SOUND_INPUT_LINE
SOUND_INPUT_CD

See also:

See Section 27.8 [start_sound_input], page 179.

27.8 start_sound_input

`int start_sound_input(int rate, int bits, int stereo);`

Starts recording in the specified format, suspending audio playback as necessary (this will always happen with the current drivers). Returns the buffer size in bytes if successful, or zero on error.

See also:

See Section 27.1 [install_sound_input], page 177.

See Section 27.10 [read_sound_input], page 180.

See Section 27.9 [stop_sound_input], page 180.

See Section 27.11 [digi_recorder], page 180.

See Section 27.7 [set_sound_input_source], page 179.

See Section 27.6 [get_sound_input_cap_parm], page 179.

See Section 27.3 [get_sound_input_cap_bits], page 178.

See Section 27.5 [get_sound_input_cap_rate], page 178.

See Section 27.4 [get_sound_input_cap_stereo], page 178.

27.9 stop_sound_input

```
void stop_sound_input();
```

Stops audio recording, switching the card back into the normal playback mode.

See also:

See Section 27.8 [start_sound_input], page 179.

27.10 read_sound_input

```
int read_sound_input(void *buffer);
```

Retrieves the most recently recorded audio buffer into the specified location, returning non-zero if a buffer has been copied or zero if no new data is yet available. The buffer size can be obtained by checking the return value from start_sound_input(). You must be sure to call this function at regular intervals during the recording (typically around 100 times a second), or some data will be lost. If you are unable to do this often enough from the mainline code, use the digi_recorder() callback to store the waveform into a larger buffer of your own. Note: many cards produce a click or popping sound when switching between record and playback modes, so it is often a good idea to discard the first buffer after you start a recording. The waveform is always stored in unsigned format, with stereo data consisting of alternate left/right samples.

See also:

See Section 27.8 [start_sound_input], page 179.

27.11 digi_recorder

```
extern void (*digi_recorder)();
```

If set, this function is called by the input driver whenever a new sample buffer becomes available, at which point you can use read_sound_input() to copy the data into a more permenent location. This routine runs in an interrupt context, so it must execute very quickly, the code and all memory that it touches must be locked, and you cannot call any operating system routines or access disk files.

See also:

See Section 27.1 [install_sound_input], page 177.

See Section 27.8 [start_sound_input], page 179.

27.12 midi_recorder

`extern void (*midi_recorder)(unsigned char data);`

If set, this function is called by the MIDI input driver whenever a new byte of MIDI data becomes available. It runs in an interrupt context, so it must execute very quickly and all the code/data must be locked.

See also:
See Section 27.1 [install_sound_input], page 177.
See Section 25.10 [midi_out], page 173.

28 File and compression routines

The following routines implement a fast buffered file I/O system, which supports the reading and writing of compressed files using a ring buffer algorithm based on the LZSS compressor by Haruhiko Okumura. This does not achieve quite such good compression as programs like zip and lha, but unpacking is very fast and it does not require much memory. Packed files always begin with the 32 bit value F_PACK_MAGIC, and autodetect files with the value F_NOPACK_MAGIC.

The following FA_* flags are guaranteed to work: FA_RDONLY, FA_HIDDEN, FA_SYSTEM, FA_LABEL, FA_DIREC, FA_ARCH. Do not use any other flags from DOS/Windows or your code will not compile on another platform. Flags FA_SYSTEM, FA_LABEL and FA_ARCH are valuable only on DOS/Windows (entries with system flag, volume labels and archive flag). FA_RDONLY is for directory entries with read-only flag on DOS-like systems or unwritable by current user on Unix-like systems. FA_HIDDEN is for entries with hidden flag on DOS-like systems or starting with '.' on Unix (dotted files - excluding '.' and '..'). FA_DIREC represents directories. Flags can be combined using '`|`' (binary OR operator).

When passed to the functions as the 'attrib' parameter, these flags represent an upper set in which the actual flag set of a matching file must be included. That is, in order for a file to be matching, its attributes may contain any of the specified flags but must not contain any of the unspecified flags. Thus, if you pass 'FA_DIREC `|` FA_RDONLY', normal files and directories will be included as well as read-only files and directories, but not hidden files and directories. Similarly, if you pass 'FA_ARCH' then both archived and non-archived files will be included.

28.1 get_executable_name

`void get_executable_name(char *buf, int size);`

Fills buf with the full path to the current executable, writing at most size bytes. This generally comes from argv[0], but on Unix systems if argv[0] does not specify the path, we search for our file in $PATH.

28.2 fix_filename_case

`char *fix_filename_case(char *path);`

Converts a filename to a standardised case. On DOS platforms, they will be entirely uppercase. Returns a copy of the path parameter.

See also:

See Section 28.3 [fix_filename_slashes], page 182.

See Section 28.4 [fix_filename_path], page 182.

28.3 fix_filename_slashes

`char *fix_filename_slashes(char *path);`

Converts all the directory separators in a filename to a standard character. On DOS platforms, this is a backslash. Returns a copy of the path parameter.

See also:

See Section 28.2 [fix_filename_case], page 182.

See Section 28.4 [fix_filename_path], page 182.

28.4 fix_filename_path

`char *fix_filename_path(char *dest, const char *path, int size);`

Converts a partial filename into a full path, storing at most size bytes into the dest buffer. Returns a copy of the dest parameter.

See also:

See Section 28.2 [fix_filename_case], page 182.

See Section 28.3 [fix_filename_slashes], page 182.

28.5 replace_filename

```
char *replace_filename(char *dest, const char *path, const char *filename,
int size);
```

Replaces the specified path+filename with a new filename tail, storing at most size bytes into the dest buffer. Returns a copy of the dest parameter.

See also:

See Section 28.8 [get_filename], page 183.

See Section 28.6 [replace_extension], page 183.

See Section 28.7 [append_filename], page 183.

28.6 replace_extension

```
char *replace_extension(char *dest, const char *filename, const char *ext,
int size);
```

Replaces the specified filename+extension with a new extension tail, storing at most size bytes into the dest buffer. Returns a copy of the dest parameter.

See also:

See Section 28.9 [get_extension], page 183.

See Section 28.5 [replace_filename], page 182.

28.7 append_filename

```
char *append_filename(char *dest, const char *path, const char *filename,
int size);
```

Concatenates the specified filename onto the end of the specified path, storing at most size bytes into the dest buffer. Returns a copy of the dest parameter.

See also:

See Section 28.5 [replace_filename], page 182.

28.8 get_filename

```
char *get_filename(const char *path);
```

When passed a completely specified file path, this returns a pointer to the filename portion. Both '\' and '/' are recognized as directory separators.

See also:

See Section 28.9 [get_extension], page 183.

See Section 28.10 [put_backslash], page 184.

See Section 28.5 [replace_filename], page 182.

28.9 get_extension

```
char *get_extension(const char *filename);
```

When passed a complete filename (with or without path information) this returns a pointer to the file extension.

See also:

See Section 28.8 [get_filename], page 183.

See Section 28.10 [put_backslash], page 184.

See Section 28.6 [replace_extension], page 183.

28.10 put_backslash

`void put_backslash(char *filename);`

If the last character of the filename is not a '\', '/', '#' or a device separator (ie. ':' under DOS), this routine will concatenate either a '\' or '/' on to it (depending on the platform). Note: ignore the function name, it's out of date.

See also:

See Section 28.9 [get_extension], page 183.

See Section 28.8 [get_filename], page 183.

28.11 file_exists

`int file_exists(const char *filename, int attrib, int *aret);`

Checks whether a file matching the given name and attributes (see above) exists, returning non-zero if it does. If aret is not NULL, it will be set to the attributes of the matching file. If an error occurs the system error code will be stored in errno.

See also:

See Section 28.12 [exists], page 184.

See Section 28.13 [file_size], page 184.

See Section 28.14 [file_time], page 185.

28.12 exists

`int exists(const char *filename);`

Shortcut version of file_exists(), which checks for normal files, which may have the archive or read-only bits set, but are not hidden, directories, system files, etc.

See also:

See Section 28.11 [file_exists], page 184.

See Section 28.13 [file_size], page 184.

See Section 28.14 [file_time], page 185.

28.13 file_size

```
long file_size(const char *filename);
```

Returns the size of a file, in bytes. If the file does not exist or an error occurs, it will return zero and store the system error code in errno.

See also:

See Section 28.11 [file_exists], page 184.

See Section 28.14 [file_time], page 185.

28.14 file_time

```
time_t file_time(const char *filename);
```

Returns the modification time (number of seconds since 00:00:00 GMT 1/1/1970) of a file. If the file does not exist or an error occurs, it will return zero and store the system error code in errno.

See also:

See Section 28.11 [file_exists], page 184.

See Section 28.13 [file_size], page 184.

28.15 delete_file

```
int delete_file(const char *filename);
```

Removes a file from the disk.

28.16 for_each_file

```
int for_each_file(const char *name, int attrib, void (*callback)(const char
*filename, int attrib, int param), int param);
```

Finds all the files on the disk which match the given wildcard specification and file attributes (see above), and executes callback() once for each. callback() will be passed three arguments, the first a string which contains the completed filename, the second being the attributes of the file, and the third an int which is simply a copy of param (you can use this for whatever you like). If an error occurs an error code will be stored in errno, and callback() can cause for_each_file() to abort by setting errno itself. Returns the number of successful calls made to callback().

28.17 al_findfirst

```
int al_findfirst(const char *pattern, struct al_ffblk *info, int attrib);
```

Low-level function for searching files. This function finds the first file which matches the given wildcard specification and file attributes (see above). The information about the file (if any) will be put in the al_ffblk structure which you have to provide. The function returns zero if a match is found, nonzero if none is found or if an error occured and, in the latter case, sets errno accordingly. The al_ffblk structure looks like:

```
struct al_ffblk
{
    int attrib;       - actual attributes of the file found
    time_t time;      - modification time of file
    long size;        - size of file
    char name[512];   - name of file
};
```

There is some other stuff in the structure as well, but it is there for internal use only.

See also:

See Section 28.18 [al_findnext], page 186.

See Section 28.19 [al_findclose], page 186.

28.18 al_findnext

`int al_findnext(struct al_ffblk *info);`

This finds the next file in a search started by al_findfirst(). Returns zero if a match is found, nonzero if none is found or if an error occured and, in the latter case, sets errno accordingly.

See also:

See Section 28.17 [al_findfirst], page 185.

See Section 28.19 [al_findclose], page 186.

28.19 al_findclose

`void al_findclose(struct al_ffblk *info);`

This closes a previously opened search with al_findfirst().

See also:

See Section 28.17 [al_findfirst], page 185.

See Section 28.18 [al_findnext], page 186.

28.20 find_allegro_resource

`int find_allegro_resource(char *dest, const char *resource, const char *ext, const char *datafile, const char *objectname, const char *envvar, const char *subdir, int size);`

Searches for a support file, eg. allegro.cfg or language.dat. Passed a resource string describing what you are looking for, along with extra optional information such as the default extension, what datafile to look inside, what the datafile object name is likely to be, any special environment variable to check, and any subdirectory that you would like to check as well as the default location, this function looks in a hell of a lot of different places :-) Returns zero on success, and stores a full path to the file (at most size bytes) into the dest buffer.

28.21 packfile_password

```
void packfile_password(const char *password);
```

Sets the encryption password to be used for all read/write operations on files opened in future using Allegro's packfile functions (whether they are compressed or not), including all the save, load and config routines. Files written with an encryption password cannot be read unless the same password is selected, so be careful: if you forget the key, I can't make your data come back again! Pass NULL or an empty string to return to the normal, non-encrypted mode. If you are using this function to prevent people getting access to your datafiles, be careful not to store an obvious copy of the password in your executable: if there are any strings like "I'm the password for the datafile", it would be fairly easy to get access to your data :-)

Note #1: when writing a packfile, you can change the password to whatever you want after opening the file, without affecting the write operation. On the contrary, when writing a sub-chunk of a packfile, you must make sure that the password that was active at the time the sub-chunk was opened is still active before closing the sub-chunk. This is guaranteed to be true if you didn't call the packfile_password() routine in the meantime. Read operations, either on packfiles or sub-chunks, have no such restriction.

Note #2: as explained above, the password is used for all read/write operations on files, including for several functions of the library that operate on files without explicitly using packfiles, e.g load_bitmap(). The unencrypted mode is mandatory in order for those functions to work. Therefore remember to call packfile_password(NULL) before using them if you previously changed the password. As a rule of thumb, always call packfile_password(NULL) when you are done with operations on packfiles.

See also:

See Section 28.22 [pack_fopen], page 187.

See Section 29.1 [load_datafile], page 191.

28.22 pack_fopen

```
PACKFILE *pack_fopen(const char *filename, const char *mode);
```

Opens a file according to mode, which may contain any of the flags:

- 'r' - open file for reading.
- 'w' - open file for writing, overwriting any existing data.
- 'p' - open file in packed mode. Data will be compressed as it is written to the file, and automatically uncompressed during read operations. Files

created in this mode will produce garbage if they are read without this flag being set.

- '!' - open file for writing in normal, unpacked mode, but add the value F_NOPACK_MAGIC to the start of the file, so that it can later be opened in packed mode and Allegro will automatically detect that the data does not need to be decompressed.

Instead of these flags, one of the constants F_READ, F_WRITE, F_READ_PACKED, F_WRITE_PACKED or F_WRITE_NOPACK may be used as the mode parameter. On success, pack_fopen() returns a pointer to a file structure, and on error it returns NULL and stores an error code in errno. An attempt to read a normal file in packed mode will cause errno to be set to EDOM.

The packfile functions also understand several "magic" filenames that are used for special purposes. These are:

- "#" - read data that has been appended to your executable file with the exedat utility, as if it was a regular independent disk file.
- 'filename.dat#object_name' - open a specific object from a datafile, and read from it as if it was a regular file. You can treat nested datafiles exactly like a normal directory structure, for example you could open 'filename.dat#graphics/level1/mapdata'.
- '#object_name' - combination of the above, reading an object from a datafile that has been appended onto your executable.

With these special filenames, the contents of a datafile object or appended file can be read in an identical way to a normal disk file, so any of the file access functions in Allegro (eg. load_pcx() and set_config_file()) can be used to read from them. Note that you can't write to these special files, though: the fake file is read only. Also, you must save your datafile uncompressed or with per-object compression if you are planning on loading individual objects from it (otherwise there will be an excessive amount of seeking when it is read). Finally, be aware that the special Allegro object types aren't the same format as the files you import the data from. When you import data like bitmaps or samples into the grabber, they are converted into a special Allegro-specific format, but the '#' marker file syntax reads the objects as raw binary chunks. This means that if, for example, you want to use load_pcx to read an image from a datafile, you should import it as a binary block rather than as a BITMAP object.

See also:
See Section 28.23 [packfile functions], page 188.
See Section 28.24 [pack_fopen_chunk], page 190.
See Section 28.21 [packfile_password], page 187.
See Section 33.45 [file_select], page 225.

28.23 packfile functions

```
int pack_fclose(PACKFILE *f);
int pack_fseek(PACKFILE *f, int offset);
int pack_feof(PACKFILE *f);
int pack_ferror(PACKFILE *f);
int pack_getc(PACKFILE *f);
int pack_putc(int c, PACKFILE *f);
int pack_igetw(PACKFILE *f);
long pack_igetl(PACKFILE *f);
int pack_iputw(int w, PACKFILE *f);
long pack_iputl(long l, PACKFILE *f);
int pack_mgetw(PACKFILE *f);
long pack_mgetl(PACKFILE *f);
int pack_mputw(int w, PACKFILE *f);
long pack_mputl(long l, PACKFILE *f);
long pack_fread(void *p, long n, PACKFILE *f);
long pack_fwrite(const void *p, long n, PACKFILE *f);
char *pack_fgets(char *p, int max, PACKFILE *f);
int pack_fputs(const char *p, PACKFILE *f);
```

These work like the equivalent stdio functions. There are some differences, however:

- Seeking only supports forward movement relative to the current position. Note that seeking is very slow when reading compressed files, and so should be avoided unless you are sure that the file is not compressed.
- The pack_i* and pack_m* routines read and write 16 and 32 bit values using the Intel and Motorola byte ordering systems (endianness) respectively. Intel is least significant byte first (little-endian); Motorola is most significant byte first (big-endian).
- pack_fread() and pack_fwrite() take a single size parameter instead of that silly size and num_elements system.
- The pack_fgets() function does not include a trailing carriage return in the returned string.
- pack_fputs() always writes in the UTF-8 text encoding format, converting from the current text encoding. Newlines (\n) are written as \r\n on DOS/Windows. If you do not want either of these things to happen, use pack_fwrite() and/or pack_putc() instead.
- pack_feof() returns nonzero as soon as you reach the end of the file. It does not wait for you to attempt to read beyond the end of the file, contrary to the ISO C feof() function. The only way to know whether you have read beyond the end of the file is to check the return value of the read operation you use (and be wary of pack_*getl() as EOF is also a valid return value with these functions).

See also:

See Section 28.22 [pack_fopen], page 187.

28.24 pack_fopen_chunk

```
PACKFILE *pack_fopen_chunk(PACKFILE *f, int pack);
```

Opens a sub-chunk of a file. Chunks are primarily intended for use by the datafile code, but they may also be useful for your own file routines. A chunk provides a logical view of part of a file, which can be compressed as an individual entity and will automatically insert and check length counts to prevent reading past the end of the chunk. To write a chunk to the file f, use the code:

```
/* assumes f is a PACKFILE * which has been opened */
f = pack_fopen_chunk(f, pack);    /* in write mode */
write some data to f
f = pack_fclose_chunk(f);
```

The data written to the chunk will be prefixed with two length counts (32 bit, big-endian). For uncompressed chunks these will both be set to the size of the data in the chunk. For compressed chunks (created by setting the pack flag), the first length will be the raw size of the chunk, and the second will be the negative size of the uncompressed data.

To read the chunk, use the code:

```
/* assumes f is a PACKFILE * which has been opened */
f = pack_fopen_chunk(f, FALSE);    */ in read mode */
read data from f
f = pack_fclose_chunk(f);
```

This sequence will read the length counts created when the chunk was written, and automatically decompress the contents of the chunk if it was compressed. The length will also be used to prevent reading past the end of the chunk (Allegro will return EOF if you attempt this), and to automatically skip past any unread chunk data when you call pack_fclose_chunk().

Chunks can be nested inside each other by making repeated calls to pack_fopen_chunk(). When writing a file, the compression status is inherited from the parent file, so you only need to set the pack flag if the parent is not compressed but you want to pack the chunk data. If the parent file is already open in packed mode, setting the pack flag will result in data being compressed twice: once as it is written to the chunk, and again as the chunk passes it on to the parent file.

See also:

See Section 28.25 [pack_fclose_chunk], page 191.

See Section 28.22 [pack_fopen], page 187.

28.25 pack_fclose_chunk

`PACKFILE *pack_fclose_chunk(PACKFILE *f);`

Closes a sub-chunk of a file, previously obtained by calling pack_fopen_chunk().

See also:
See Section 28.24 [pack_fopen_chunk], page 190.

29 Datafile routines

Datafiles are created by the grabber utility, and have a .dat extension. They can contain bitmaps, palettes, fonts, samples, MIDI music, FLI/FLC animations, and any other binary data that you import.

Warning: when using truecolor images, you should always set the graphics mode before loading any bitmap data! Otherwise the pixel format (RGB or BGR) will not be known, so the file may be converted wrongly.

See the documentation for pack_fopen() for information about how to read directly from a specific datafile object.

29.1 load_datafile

`DATAFILE *load_datafile(const char *filename);`

Loads a datafile into memory, and returns a pointer to it, or NULL on error. If the datafile has been encrypted, you must first use the packfile_password() function to set the appropriate key. See grabber.txt for more information. If the datafile contains truecolor graphics, you must set the video mode or call set_color_conversion() before loading it.

See also:

See Section 29.2 [load_datafile_callback], page 191.

See Section 29.3 [unload_datafile], page 192.

See Section 29.4 [load_datafile_object], page 192.

See Section 10.11 [set_color_conversion], page 90.

See Section 29.9 [fixup_datafile], page 193.

See Section 28.21 [packfile_password], page 187.

See Section 29.6 [find_datafile_object], page 193.

See Section 29.8 [register_datafile_object], page 193.

29.2 load_datafile_callback

```
DATAFILE *load_datafile_callback(const char *filename, void
(*callback)(DATAFILE *d));
```

Loads a datafile into memory, calling the specified hook function once for each object in the file, passing it a pointer to the object just read.

See also:
See Section 29.1 [load_datafile], page 191.
See Section 29.3 [unload_datafile], page 192.
See Section 29.4 [load_datafile_object], page 192.
See Section 10.11 [set_color_conversion], page 90.
See Section 29.9 [fixup_datafile], page 193.
See Section 28.21 [packfile_password], page 187.
See Section 29.6 [find_datafile_object], page 193.
See Section 29.8 [register_datafile_object], page 193.

29.3 unload_datafile

`void unload_datafile(DATAFILE *dat);`

Frees all the objects in a datafile.

See also:
See Section 29.1 [load_datafile], page 191.

29.4 load_datafile_object

```
DATAFILE *load_datafile_object(const char *filename, const char
*objectname);
```

Loads a specific object from a datafile. This won't work if you strip the object names from the file, and it will be very slow if you save the file with global compression. See grabber.txt for more information.

See also:
See Section 29.5 [unload_datafile_object], page 192.
See Section 29.1 [load_datafile], page 191.
See Section 10.11 [set_color_conversion], page 90.
See Section 29.6 [find_datafile_object], page 193.
See Section 29.8 [register_datafile_object], page 193.

29.5 unload_datafile_object

`void unload_datafile_object(DATAFILE *dat);`

Frees an object previously loaded by load_datafile_object().

See also:
See Section 29.4 [load_datafile_object], page 192.

29.6 find_datafile_object

```
DATAFILE *find_datafile_object(const DATAFILE *dat, const char
*objectname);
```

Searches an already loaded datafile for an object with the specified name, returning a pointer to it, or NULL if the object cannot be found. It understands '/' and '#' separators for nested datafile paths.

See also:

See Section 29.1 [load_datafile], page 191.

See Section 29.4 [load_datafile_object], page 192.

29.7 get_datafile_property

```
const char *get_datafile_property(const DATAFILE *dat, int type);
```

Returns the specified property string for the object, or an empty string if the property isn't present. See grabber.txt for more information.

29.8 register_datafile_object

```
void register_datafile_object(int id, void *(*load)(PACKFILE *f, long
size), void (*destroy)(void *data));
```

Used to add custom object types, specifying functions to load and destroy objects of this type. See grabber.txt for more information.

See also:

See Section 29.1 [load_datafile], page 191.

See Section 29.4 [load_datafile_object], page 192.

29.9 fixup_datafile

```
void fixup_datafile(DATAFILE *data);
```

If you are using compiled datafiles (produced by the dat2s utility) on a platform that doesn't support constructors, or on a platform that does support constructors and the datafiles contain truecolor images, you must call this function once after your set the video mode that you will be using. This will ensure the datafiles are properly initialised in the first case and convert the color values into the appropriate format in the second case. It handles flipping between RGB and BGR formats, and converting between different color depths whenever that can be done without changing the size of the image (ie. changing between 15<->16 bit hicolor for both bitmaps and RLE sprites, and 24<->32 bit truecolor for RLE sprites).

See also:

See Section 8.6 [set_gfx_mode], page 70.

See Section 10.11 [set_color_conversion], page 90.

29.10 using datafiles

When you load a datafile, you will obtain a pointer to an array of DATAFILE structures:

```
typedef struct DATAFILE
{
   void *dat;     - pointer to the actual data
   int type;      - type of the data
   long size;     - size of the data in bytes
   void *prop;    - list of object properties
} DATAFILE;
```

The type field will be one of the values:

```
DAT_FILE       - dat points to a nested datafile
DAT_DATA       - dat points to a block of binary data
DAT_FONT       - dat points to a font object
DAT_SAMPLE     - dat points to a sample structure
DAT_MIDI       - dat points to a MIDI file
DAT_PATCH      - dat points to a GUS patch file
DAT_FLI        - dat points to an FLI/FLC animation
DAT_BITMAP     - dat points to a BITMAP structure
DAT_RLE_SPRITE - dat points to a RLE_SPRITE structure
DAT_C_SPRITE   - dat points to a linear compiled sprite
DAT_XC_SPRITE  - dat points to a mode-X compiled sprite
DAT_PALETTE    - dat points to an array of 256 RGB structures
DAT_END        - special flag to mark the end of the data list
```

The grabber program can also produce a header file defining the index of each object within the file as a series of #defined constants, using the names you gave the objects in the grabber. So, for example, if you have made a datafile called foo.dat which contains a bitmap called THE_IMAGE, you could display it with the code fragment:

```
#include "foo.h"

DATAFILE *data = load_datafile("foo.dat");
draw_sprite(screen, data[THE_IMAGE].dat, x, y);
```

If you are programming in C++ you will get an error because the dat field is a void pointer and draw_sprite() expects a BITMAP pointer. You can get around this with a cast, eg:

```
draw_sprite(screen, (BITMAP *)data[THE_IMAGE].dat, x, y);
```

When you load a single datafile object, you will obtain a pointer to a single DATAFILE structure. This means that you don't access it any more like an array, and it doesn't have any DAT_END object. Example:

```
     music_object = load_datafile_object("datafile.dat", "MUSIC");
     play_midi(music_object->dat);
```

30 Fixed point math routines

Allegro provides some routines for working with fixed point numbers, and defines the type 'fixed' to be a signed 32 bit integer. The high word is used for the integer part and the low word for the fraction, giving a range of -32768 to 32767 and an accuracy of about four or five decimal places. Fixed point numbers can be assigned, compared, added, subtracted, negated and shifted (for multiplying or dividing by powers of two) using the normal integer operators, but you should take care to use the appropriate conversion routines when mixing fixed point with integer or floating point values. Writing 'fixed_point_1 + fixed_point_2' is ok, but 'fixed_point + integer' is not.

30.1 itofix

`fixed itofix(int x);`

Converts an integer to fixed point. This is the same thing as x<<16.

See also:

See Section 30.2 [fixtoi], page 195.

See Section 30.5 [ftofix], page 196.

See Section 30.6 [fixtof], page 196.

30.2 fixtoi

`int fixtoi(fixed x);`

Converts fixed point to integer, rounding as required.

See also:

See Section 30.1 [itofix], page 195.

See Section 30.5 [ftofix], page 196.

See Section 30.6 [fixtof], page 196.

See Section 30.3 [fixfloor], page 195.

See Section 30.4 [fixceil], page 196.

30.3 fixfloor

`int fixfloor(fixed x);`

Returns the greatest integer not greater than x. That is, it rounds towards negative infinity.

See also:

See Section 30.2 [fixtoi], page 195.

See Section 30.4 [fixceil], page 196.

30.4 fixceil

`int fixceil(fixed x);`

Returns the smallest integer not less than x. That is, it rounds towards positive infinity.

See also:

See Section 30.2 [fixtoi], page 195.

See Section 30.3 [fixfloor], page 195.

30.5 ftofix

`fixed ftofix(double x);`

Converts a floating point value to fixed point.

See also:

See Section 30.6 [fixtof], page 196.

See Section 30.1 [itofix], page 195.

See Section 30.2 [fixtoi], page 195.

30.6 fixtof

`double fixtof(fixed x);`

Converts fixed point to floating point.

See also:

See Section 30.5 [ftofix], page 196.

See Section 30.1 [itofix], page 195.

See Section 30.2 [fixtoi], page 195.

30.7 fixmul

`fixed fixmul(fixed x, fixed y);`

A fixed point value can be multiplied or divided by an integer with the normal '*' and '/' operators. To multiply two fixed point values, though, you must use this function.

If an overflow or division by zero occurs, errno will be set and the maximum possible value will be returned, but errno is not cleared if the operation is successful. This means that if you are going to test for overflow you should set errno=0 before calling fixmul().

See also:

See Section 30.9 [fixadd], page 197.

See Section 30.10 [fixsub], page 197.

See Section 30.8 [fixdiv], page 197.

30.8 fixdiv

`fixed fixdiv(fixed x, fixed y);`

Fixed point division: see comments about fixmul().

See also:

See Section 30.9 [fixadd], page 197.

See Section 30.10 [fixsub], page 197.

See Section 30.7 [fixmul], page 196.

30.9 fixadd

`fixed fixadd(fixed x, fixed y);`

Although fixed point numbers can be added with the normal '`+`' integer operator, that doesn't provide any protection against overflow. If overflow is a problem, you should use this function instead. It is slower than using integer operators, but if an overflow occurs it will clamp the result, rather than just letting it wrap, and set errno.

See also:

See Section 30.10 [fixsub], page 197.

See Section 30.7 [fixmul], page 196.

See Section 30.8 [fixdiv], page 197.

30.10 fixsub

`fixed fixsub(fixed x, fixed y);`

Fixed point subtraction: see comments about fixadd().

See also:

See Section 30.9 [fixadd], page 197.

See Section 30.7 [fixmul], page 196.

See Section 30.8 [fixdiv], page 197.

30.11 fixed point trig

The fixed point square root, sin, cos, tan, inverse sin, and inverse cos functions are implemented using lookup tables, which are very fast but not particularly accurate. At the moment the inverse tan uses an iterative search on the tan table, so it is a lot slower than the others.

Angles are represented in a binary format with 256 equal to a full circle, 64 being a right angle and so on. This has the advantage that a simple bitwise 'and' can be used to keep the angle within the range zero to a full circle, eliminating all those tiresome 'if (angle >= 360)' checks.

30.12 fixsin

```
fixed fixsin(fixed x);
```

Lookup table sine.

See also:

See Section 30.11 [fixed point trig], page 197.

30.13 fixcos

```
fixed fixcos(fixed x);
```

Lookup table cosine.

See also:

See Section 30.11 [fixed point trig], page 197.

30.14 fixtan

```
fixed fixtan(fixed x);
```

Lookup table tangent.

See also:

See Section 30.11 [fixed point trig], page 197.

30.15 fixasin

```
fixed fixasin(fixed x);
```

Lookup table inverse sine.

See also:

See Section 30.11 [fixed point trig], page 197.

30.16 fixacos

```
fixed fixacos(fixed x);
```

Lookup table inverse cosine.

See also:

See Section 30.11 [fixed point trig], page 197.

30.17 fixatan

`fixed fixatan(fixed x);`

Fixed point inverse tangent.

See also:

See Section 30.11 [fixed point trig], page 197.

30.18 fixatan2

`fixed fixatan2(fixed y, fixed x);`

Fixed point version of the libc atan2() routine.

See also:

See Section 30.11 [fixed point trig], page 197.

30.19 fixsqrt

`fixed fixsqrt(fixed x);`

Fixed point square root.

30.20 fixhypot

`fixed fixhypot(fixed x, fixed y);`

Fixed point hypotenuse (returns the square root of x*x + y*y).

30.21 fix class

If you are programming in C++ you can ignore all the above and use the fix class instead, which overloads a lot of operators to provide automatic conversion to and from integer and floating point values, and calls the above routines as they are required. You should not mix the fix class with the fixed typedef though, because the compiler will mistake the fixed values for regular integers and insert unnecessary conversions. For example, if x is an object of class fix, calling fixsqrt(x) will return the wrong result. You should use the overloaded sqrt(x) or x.sqrt() instead.

30.22 fixed point aliases

The fixed point functions used to be named with an "f" prefix instead of "fix", eg. fixsqrt() used to be fsqrt(), but were renamed due to conflicts with some libc implementations. This should not affect most existing code as there are backwards compatibility aliases. These aliases are static inline functions which map the old names to the new names, eg. fsqrt() calls fixsqrt(). You can disable the aliases by defining the preprocessor macro ALLEGRO_NO_FIX_ALIASES before including allegro.h.

31 3D math routines

Allegro contains some 3d helper functions for manipulating vectors, constructing and using transformation matrices, and doing perspective projections from 3d space onto the screen. It is not, and never will be, a fully fledged 3d library (my goal is to supply generic support routines, not shrink-wrapped graphics code :-) but these functions may be useful for developing your own 3d code.

Allegro uses a right-handed coordinate system, i.e. if you point the thumb of your right hand along the x axis, and the index finger along the y axis, your middle finger points in the direction of the z axis. This also means, for any rotation, if you point the thumb of your right hand along the axis of rotation, then the fingers curl in the positive direction of rotation.

All the 3d math functions are available in two versions: one which uses fixed point arithmetic, and another which uses floating point. The syntax for these is identical, but the floating point functions and structures are postfixed with '_f', eg. the fixed point function cross_product() has a floating point equivalent cross_product_f(). If you are programming in C++, Allegro also overloads these functions for use with the 'fix' class.

3d transformations are accomplished by the use of a modelling matrix. This is a 4x4 array of numbers that can be multiplied with a 3d point to produce a different 3d point. By putting the right values into the matrix, it can be made to do various operations like translation, rotation, and scaling. The clever bit is that you can multiply two matrices together to produce a third matrix, and this will have the same effect on points as applying the original two matrices one after the other. For example, if you have one matrix that rotates a point and another that shifts it sideways, you can combine them to produce a matrix that will do the rotation and the shift in a single step. You can build up extremely complex transformations in this way, while only ever having to multiply each point by a single matrix.

Allegro actually cheats in the way it implements the matrix structure. Rotation and scaling of a 3d point can be done with a simple 3x3 matrix, but in order to translate it and project it onto the screen, the matrix must be extended to 4x4, and the point extended into 4d space by the addition of an extra coordinate, w=1. This is a bad thing in terms of efficiency, but fortunately an optimisation is possible. Given the 4x4 matrix:

```
   ( a, b, c, d )
   ( e, f, g, h )
   ( i, j, k, l )
   ( m, n, o, p )
```

a pattern can be observed in which parts of it do what. The top left 3x3 grid implements rotation and scaling. The three values in the top right column (d, h, and l) implement translation, and as long as the matrix is only used for affine transformations, m, n and o will always be zero and p will always be 1. If you don't know what affine means, read Foley & Van Damme: basically it covers scaling, translation, and rotation, but not projection. Since Allegro uses a separate function for projection, the matrix functions only need to support affine transformations, which means that there is no need to store the bottom row of the matrix. Allegro implicitly assumes that it contains (0,0,0,1), and optimises the matrix manipulation functions accordingly.

Matrices are stored in the structures:

```
typedef struct MATRIX            - fixed point matrix structure
{
   fixed v[3][3];                - 3x3 scaling and rotation component
   fixed t[3];                   - x/y/z translation component
} MATRIX;

typedef struct MATRIX_f          - floating point matrix structure
{
   float v[3][3];                - 3x3 scaling and rotation component
   float t[3];                   - x/y/z translation component
} MATRIX_f
```

31.1 identity_matrix

`extern MATRIX identity_matrix;`

`extern MATRIX_f identity_matrix_f;`

Global variables containing the 'do nothing' identity matrix. Multiplying by the identity matrix has no effect.

31.2 get_translation_matrix

`void get_translation_matrix(MATRIX *m, fixed x, fixed y, fixed z);`

`void get_translation_matrix_f(MATRIX_f *m, float x, float y, float z);`

Constructs a translation matrix, storing it in m. When applied to the point (px, py, pz), this matrix will produce the point (px+x, py+y, pz+z). In other words, it moves things sideways.

See also:

See Section 31.23 [apply_matrix], page 207.

See Section 31.11 [get_transformation_matrix], page 204.

See Section 31.15 [qtranslate_matrix], page 205.

31.3 get_scaling_matrix

`void get_scaling_matrix(MATRIX *m, fixed x, fixed y, fixed z);`

`void get_scaling_matrix_f(MATRIX_f *m, float x, float y, float z);`

Constructs a scaling matrix, storing it in m. When applied to the point (px, py, pz), this matrix will produce the point (px*x, py*y, pz*z). In other words, it stretches or shrinks things.

See also:

See Section 31.23 [apply_matrix], page 207.

See Section 31.11 [get_transformation_matrix], page 204.

See Section 31.16 [qscale_matrix], page 205.

31.4 get_x_rotate_matrix

`void get_x_rotate_matrix(MATRIX *m, fixed r);`

`void get_x_rotate_matrix_f(MATRIX_f *m, float r);`

Construct X axis rotation matrices, storing them in m. When applied to a point, these matrices will rotate it about the X axis by the specified angle (given in binary, 256 degrees to a circle format).

See also:

See Section 31.23 [apply_matrix], page 207.

See Section 31.7 [get_rotation_matrix], page 203.

See Section 31.5 [get_y_rotate_matrix], page 202.

See Section 31.6 [get_z_rotate_matrix], page 202.

31.5 get_y_rotate_matrix

`void get_y_rotate_matrix(MATRIX *m, fixed r);`

`void get_y_rotate_matrix_f(MATRIX_f *m, float r);`

Construct Y axis rotation matrices, storing them in m. When applied to a point, these matrices will rotate it about the Y axis by the specified angle (given in binary, 256 degrees to a circle format).

See also:

See Section 31.23 [apply_matrix], page 207.

See Section 31.7 [get_rotation_matrix], page 203.

See Section 31.4 [get_x_rotate_matrix], page 202.

See Section 31.6 [get_z_rotate_matrix], page 202.

31.6 get_z_rotate_matrix

`void get_z_rotate_matrix(MATRIX *m, fixed r);`

`void get_z_rotate_matrix_f(MATRIX_f *m, float r);`

Construct Z axis rotation matrices, storing them in m. When applied to a point, these matrices will rotate it about the Z axis by the specified angle (given in binary, 256 degrees to a circle format).

See also:

See Section 31.23 [apply_matrix], page 207.

See Section 31.7 [get_rotation_matrix], page 203.

See Section 31.4 [get_x_rotate_matrix], page 202.

See Section 31.5 [get_y_rotate_matrix], page 202.

31.7 get_rotation_matrix

`void get_rotation_matrix(MATRIX *m, fixed x, fixed y, fixed z);`

`void get_rotation_matrix_f(MATRIX_f *m, float x, float y, float z);`

Constructs a transformation matrix which will rotate points around all three axis by the specified amounts (given in binary, 256 degrees to a circle format).

See also:

See Section 31.23 [apply_matrix], page 207.

See Section 31.11 [get_transformation_matrix], page 204.

See Section 31.10 [get_vector_rotation_matrix], page 203.

See Section 31.4 [get_x_rotate_matrix], page 202.

See Section 31.5 [get_y_rotate_matrix], page 202.

See Section 31.6 [get_z_rotate_matrix], page 202.

See Section 31.8 [get_align_matrix], page 203.

31.8 get_align_matrix

`void get_align_matrix(MATRIX *m, fixed xfront, yfront, zfront, fixed xup, fixed yup, fixed zup);`

Rotates a matrix so that it is aligned along the specified coordinate vectors (they need not be normalized or perpendicular, but the up and front must not be equal). A front vector of 1,0,0 and up vector of 0,1,0 will return the identity matrix.

See also:

See Section 31.23 [apply_matrix], page 207.

See Section 31.13 [get_camera_matrix], page 204.

31.9 get_align_matrix_f

`void get_align_matrix_f(MATRIX *m, float xfront, yfront, zfront, float xup, yup, zup);`

Floating point version of get_align_matrix().

See also:

See Section 31.8 [get_align_matrix], page 203.

31.10 get_vector_rotation_matrix

`void get_vector_rotation_matrix(MATRIX *m, fixed x, y, z, fixed a);`

`void get_vector_rotation_matrix_f(MATRIX_f *m, float x, y, z, float a);`

Constructs a transformation matrix which will rotate points around the specified x,y,z vector by the specified angle (given in binary, 256 degrees to a circle format).

See also:

See Section 31.23 [apply_matrix], page 207.

See Section 31.7 [get_rotation_matrix], page 203.

See Section 31.8 [get_align_matrix], page 203.

31.11 get_transformation_matrix

`void get_transformation_matrix(MATRIX *m, fixed scale, fixed xrot, yrot, zrot, x, y, z);`

Constructs a transformation matrix which will rotate points around all three axis by the specified amounts (given in binary, 256 degrees to a circle format), scale the result by the specified amount (pass 1 for no change of scale), and then translate to the requested x, y, z position.

See also:

See Section 31.23 [apply_matrix], page 207.

See Section 31.7 [get_rotation_matrix], page 203.

See Section 31.3 [get_scaling_matrix], page 201.

See Section 31.2 [get_translation_matrix], page 201.

31.12 get_transformation_matrix_f

`void get_transformation_matrix_f(MATRIX_f *m, float scale, float xrot, yrot, zrot, x, y, z);`

Floating point version of get_transformation_matrix().

See also:

See Section 31.11 [get_transformation_matrix], page 204.

31.13 get_camera_matrix

`void get_camera_matrix(MATRIX *m, fixed x, y, z, xfront, yfront, zfront, fixed xup, yup, zup, fov, aspect);`

Constructs a camera matrix for translating world-space objects into a normalised view space, ready for the perspective projection. The x, y, and z parameters specify the camera position, xfront, yfront, and zfront are the 'in front' vector specifying which way the camera is facing (this can be any length: normalisation is not required), and xup, yup, and zup are the 'up' direction vector. The fov parameter specifies the field of view (ie. width of the camera focus) in binary, 256 degrees to the circle format. For typical projections, a field of view in the region 32-48 will work well. Finally, the aspect ratio is used to scale the Y dimensions of the image relative to the X axis, so you can use it to adjust the proportions of the output image (set it to 1 for no scaling).

See also:

See Section 31.23 [apply_matrix], page 207.

See Section 31.8 [get_align_matrix], page 203.

31.14 get_camera_matrix_f

```
void get_camera_matrix_f(MATRIX_f *m, float x, y, z, xfront, yfront,
zfront, float xup, yup, zup, fov, aspect);
```

Floating point version of get_camera_matrix().

See also:

See Section 31.13 [get_camera_matrix], page 204.

31.15 qtranslate_matrix

```
void qtranslate_matrix(MATRIX *m, fixed x, fixed y, fixed z);
```

```
void qtranslate_matrix_f(MATRIX_f *m, float x, float y, float z);
```

Optimised routine for translating an already generated matrix: this simply adds in the translation offset, so there is no need to build two temporary matrices and then multiply them together.

See also:

See Section 31.2 [get_translation_matrix], page 201.

31.16 qscale_matrix

```
void qscale_matrix(MATRIX *m, fixed scale);
```

```
void qscale_matrix_f(MATRIX_f *m, float scale);
```

Optimised routine for scaling an already generated matrix: this simply adds in the scale factor, so there is no need to build two temporary matrices and then multiply them together.

See also:

See Section 31.3 [get_scaling_matrix], page 201.

31.17 matrix_mul

`void matrix_mul(const MATRIX *m1, *m2, MATRIX *out);`

`void matrix_mul_f(const MATRIX_f *m1, *m2, MATRIX_f *out);`

Multiplies two matrices, storing the result in out (this may be a duplicate of one of the input matrices, but it is faster when the inputs and output are all different). The resulting matrix will have the same effect as the combination of m1 and m2, ie. when applied to a point p, (p * out) = ((p * m1) * m2). Any number of transformations can be concatenated in this way. Note that matrix multiplication is not commutative, ie. matrix_mul(m1, m2) != matrix_mul(m2, m1).

See also:

See Section 31.23 [apply_matrix], page 207.

31.18 vector_length

`fixed vector_length(fixed x, fixed y, fixed z);`

`float vector_length_f(float x, float y, float z);`

Calculates the length of the vector (x, y, z), using that good 'ole Pythagoras theorem.

See also:

See Section 31.19 [normalize_vector], page 206.

31.19 normalize_vector

`void normalize_vector(fixed *x, fixed *y, fixed *z);`

`void normalize_vector_f(float *x, float *y, float *z);`

Converts the vector (*x, *y, *z) to a unit vector. This points in the same direction as the original vector, but has a length of one.

See also:

See Section 31.18 [vector_length], page 206.

See Section 31.20 [dot_product], page 206.

See Section 31.21 [cross_product], page 207.

31.20 dot_product

`fixed dot_product(fixed x1, y1, z1, x2, y2, z2);`

`float dot_product_f(float x1, y1, z1, x2, y2, z2);`

Calculates the dot product (x1, y1, z1) . (x2, y2, z2), returning the result.

See also:

See Section 31.21 [cross_product], page 207.

See Section 31.19 [normalize_vector], page 206.

31.21 cross_product

`void cross_product(fixed x1, y1, z1, x2, y2, z2, *xout, *yout, *zout);`

`void cross_product_f(float x1, y1, z1, x2, y2, z2, *xout, *yout, *zout);`

Calculates the cross product (x1, y1, z1) x (x2, y2, z2), storing the result in (*xout, *yout, *zout). The cross product is perpendicular to both of the input vectors, so it can be used to generate polygon normals.

See also:

See Section 31.20 [dot_product], page 206.

See Section 31.22 [polygon_z_normal], page 207.

See Section 31.19 [normalize_vector], page 206.

31.22 polygon_z_normal

`fixed polygon_z_normal(const V3D *v1, *v2, *v3);`

`float polygon_z_normal_f(const V3D_f *v1, *v2, *v3);`

Finds the Z component of the normal vector to the specified three vertices (which must be part of a convex polygon). This is used mainly in back-face culling. The back-faces of closed polyhedra are never visible to the viewer, therefore they never need to be drawn. This can cull on average half the polygons from a scene. If the normal is negative the polygon can safely be culled. If it is zero, the polygon is perpendicular to the screen.

See also:

See Section 31.21 [cross_product], page 207.

31.23 apply_matrix

`void apply_matrix(const MATRIX *m, fixed x, y, z, *xout, *yout, *zout);`

`void apply_matrix_f(const MATRIX_f *m, float x, y, z, *xout, *yout, *zout);`

Multiplies the point (x, y, z) by the transformation matrix m, storing the result in (*xout, *yout, *zout).

See also:

See Section 31.17 [matrix_mul], page 206.

31.24 set_projection_viewport

```
void set_projection_viewport(int x, int y, int w, int h);
```

Sets the viewport used to scale the output of the persp_project() function. Pass the dimensions of the screen area you want to draw onto, which will typically be 0, 0, SCREEN_W, and SCREEN_H.

See also:
See Section 31.25 [persp_project], page 208.

31.25 persp_project

```
void persp_project(fixed x, y, z, *xout, *yout);
void persp_project_f(float x, y, z, *xout, *yout);
```

Projects the 3d point (x, y, z) into 2d screen space, storing the result in (*xout, *yout) and using the scaling parameters previously set by calling set_projection_viewport(). This function projects from the normalized viewing pyramid, which has a camera at the origin and facing along the positive z axis. The x axis runs left/right, y runs up/down, and z increases with depth into the screen. The camera has a 90 degree field of view, ie. points on the planes x=z and -x=z will map onto the left and right edges of the screen, and the planes y=z and -y=z map to the top and bottom of the screen. If you want a different field of view or camera location, you should transform all your objects with an appropriate viewing matrix, eg. to get the effect of panning the camera 10 degrees to the left, rotate all your objects 10 degrees to the right.

See also:
See Section 31.24 [set_projection_viewport], page 207.

32 Quaternion math routines

Quaternions are an alternate way to represent the rotation part of a transformation, and can be easier to manipulate than matrices. As with a matrix, you can encode a geometric transformations in one, concatenate several of them to merge multiple transformations, and apply them to a vector, but they can only store pure rotations. The big advantage is that you can accurately interpolate between two quaternions to get a part-way rotation, avoiding the gimbal problems of the more conventional euler angle interpolation.

Quaternions only have floating point versions, without any _f suffix. Other than that, most of the quaternion functions correspond with a matrix function that performs a similar operation.

Quaternion means 'of four parts', and that's exactly what it is. Here is the structure:

```
     typedef struct QUAT
     {
        float w, x, y, z;
     }
```

You will have lots of fun figuring out what these numbers actually mean, but that is beyond the scope of this documentation. Quaternions do work – trust me.

32.1 identity_quat

`extern QUAT identity_quat;`

Global variable containing the 'do nothing' identity quaternion. Multiplying by the identity quaternion has no effect.

32.2 get_x_rotate_quat

`void get_x_rotate_quat(QUAT *q, float r);`

`void get_y_rotate_quat(QUAT *q, float r);`

`void get_z_rotate_quat(QUAT *q, float r);`

Construct axis rotation quaternions, storing them in q. When applied to a point, these quaternions will rotate it about the relevant axis by the specified angle (given in binary, 256 degrees to a circle format).

32.3 get_rotation_quat

`void get_rotation_quat(QUAT *q, float x, float y, float z);`

Constructs a quaternion that will rotate points around all three axis by the specified amounts (given in binary, 256 degrees to a circle format).

32.4 get_vector_rotation_quat

`void get_vector_rotation_quat(QUAT *q, float x, y, z, float a);`

Constructs a quaternion that will rotate points around the specified x,y,z vector by the specified angle (given in binary, 256 degrees to a circle format).

32.5 quat_to_matrix

`void quat_to_matrix(const QUAT *q, MATRIX_f *m);`

Constructs a rotation matrix from a quaternion.

32.6 matrix_to_quat

`void matrix_to_quat(const MATRIX_f *m, QUAT *q);`

Constructs a quaternion from a rotation matrix. Translation is discarded during the conversion. Use get_align_matrix_f() if the matrix is not orthonormalized, because strange things may happen otherwise.

32.7 quat_mul

`void quat_mul(const QUAT *p, const QUAT *q, QUAT *out);`

Multiplies two quaternions, storing the result in out. The resulting quaternion will have the same effect as the combination of p and q, ie. when applied to a point, (point * out) = ((point * p) * q). Any number of rotations can be concatenated in this way. Note that quaternion multiplication is not commutative, ie. quat_mul(p, q) != quat_mul(q, p).

32.8 apply_quat

`void apply_quat(const QUAT *q, float x, y, z, *xout, *yout, *zout);`

Multiplies the point (x, y, z) by the quaternion q, storing the result in (*xout, *yout, *zout). This is quite a bit slower than apply_matrix_f(), so only use it to translate a few points. If you have many points, it is much more efficient to call quat_to_matrix() and then use apply_matrix_f().

32.9 quat_interpolate

`void quat_interpolate(const QUAT *from, *to, float t, QUAT *out);`

Constructs a quaternion that represents a rotation between from and to. The argument t can be anything between 0 and 1 and represents where between from and to the result will be. 0 returns from, 1 returns to, and 0.5 will return a rotation exactly in between. The result is copied to out. This function will create the short rotation (less than 180 degrees) between from and to.

32.10 quat_slerp

`void quat_slerp(const QUAT *from, *to, float t, QUAT *out, int how);`

The same as quat_interpolate(), but allows more control over how the rotation is done. The how parameter can be any one of the values:

QUAT_SHORT - like quat_interpolate(), use shortest path
QUAT_LONG - rotation will be greater than 180 degrees
QUAT_CW - rotate clockwise when viewed from above
QUAT_CCW - rotate counterclockwise when viewed from above
QUAT_USER - the quaternions are interpolated exactly as given

33 GUI routines

Allegro contains an object-oriented dialog manager, which was originally based on the Atari GEM system (form_do(), objc_draw(), etc: old ST programmers will know what I'm talking about :-) You can use the GUI as-is to knock out simple interfaces for things like the test program and grabber utility, or you can use it as a basis for more complicated systems of your own. Allegro lets you define your own object types by writing new dialog procedures, so you can take complete control over the visual aspects of the interface while still using Allegro to handle input from the mouse, keyboard, joystick, etc.

A GUI dialog is stored as an array of DIALOG objects, each one containing the fields:

```
typedef struct DIALOG
{
   int (*proc)(int, DIALOG *, int); - dialog procedure (message handler)
   int x, y, w, h;                  - position and size of the object
```

```
      int fg, bg;                        - foreground and background colors
      int key;                           - ASCII keyboard shortcut
      int flags;                         - flags about the status of
                                           the object
      int d1, d2;                        - whatever you want to use them for
      void *dp, *dp2, *dp3;              - pointers to more
                                           object-specific data
   } DIALOG;
```

The array should end with an object which has the proc pointer set to NULL.

The flags field may contain any combination of the bit flags:

```
   D_EXIT          - this object should close the dialog when it is clicked
   D_SELECTED      - this object is selected
   D_GOTFOCUS      - this object has got the input focus
   D_GOTMOUSE      - the mouse is currently on top of this object
   D_HIDDEN        - this object is hidden and inactive
   D_DISABLED      - this object is greyed-out and inactive
   D_DIRTY         - this object needs to be redrawn
   D_INTERNAL      - don't use this! It is for internal use by the
                     library...
   D_USER          - any powers of two above this are free for your own use
```

Each object is controlled by a dialog procedure, which is stored in the proc pointer. This will be called by the dialog manager whenever any action concerning the object is required, or you can call it directly with the object_message() function. The dialog procedure should follow the form:

```
   int foo(int msg, DIALOG *d, int c);
```

It will be passed a flag (msg) indicating what action it should perform, a pointer to the object concerned (d), and if msg is MSG_CHAR or MSG_XCHAR, the key that was pressed (c). Note that d is a pointer to a specific object, and not to the entire dialog.

The dialog procedure should return one of the values:

```
   D_O_K          - normal return status
   D_CLOSE        - tells the dialog manager to close the dialog
   D_REDRAW       - tells the dialog manager to redraw the entire dialog
   D_REDRAWME     - tells the dialog manager to redraw the current object
   D_WANTFOCUS    - requests that the input focus be given to this object
   D_USED_CHAR    - MSG_CHAR and MSG_XCHAR return this if they used the key
```

Dialog procedures may be called with any of the messages:

MSG_START:
Tells the object to initialise itself. The dialog manager sends this to all the objects in a dialog just before it displays the dialog.

MSG_END:
Sent to all objects when closing a dialog, allowing them to perform whatever cleanup operations they require.

MSG_DRAW:
Tells the object to draw itself onto the screen. The mouse pointer will be turned off when this message is sent, so the drawing code does not need to worry about it.

MSG_CLICK:
Informs the object that a mouse button has been clicked while the mouse was on top of the object. Typically an object will perform its own mouse tracking as long as the button is held down, and only return from this message handler when it is released.

MSG_DCLICK:
Sent when the user double-clicks on an object. A MSG_CLICK will be sent when the button is first pressed, then MSG_DCLICK if it is released and pressed again within a short space of time.

MSG_KEY:
Sent when the keyboard shortcut for the object is pressed, or if enter, space, or a joystick button is pressed while it has the input focus.

MSG_CHAR:
When a key is pressed, this message is sent to the object that has the input focus, with a readkey() format character code (ASCII value in the low byte, scancode in the high byte) as the c parameter. If the object deals with the keypress it should return D_USED_CHAR, otherwise it should return D_O_K to allow the default keyboard interface to operate. If you need to access Unicode character input, you should use MSG_UCHAR instead of MSG_CHAR.

MSG_UCHAR:
If an object ignores the MSG_CHAR input, this message will be sent immediately after it, passed the full Unicode key value as the c parameter. This enables you to read character codes greater than 255, but cannot tell you anything about the scancode: if you need to know that, use MSG_CHAR instead. This handler should return D_USED_CHAR if it processed the input, or D_O_K otherwise.

MSG_XCHAR:
When a key is pressed, Allegro will send a MSG_CHAR and MSG_UCHAR to the object with the input focus. If this object doesn't process the key (ie. it returns D_O_K rather than D_USED_CHAR), the dialog manager will look for an object with a matching keyboard shortcut in the key field, and send it a MSG_KEY. If this fails, it broadcasts a MSG_XCHAR to all objects in the dialog, allowing them to respond to special keypresses even when they don't have the input focus. Normally you should ignore this message (return D_O_K rather than D_USED_CHAR), in which case Allegro will perform default actions such as moving the focus in response to the arrow keys and closing the dialog if ESC is pressed.

MSG_WANTFOCUS:
Queries whether an object is willing to accept the input focus. It should return D_WANTFOCUS if it does, or D_O_K if it isn't interested in getting user input.

MSG_GOTFOCUS:
MSG_LOSTFOCUS:
Sent whenever an object gains or loses the input focus. These messages will always be followed by a MSG_DRAW, to let objects display themselves differently when they have the input focus. If you return D_WANTFOCUS in response to a MSG_LOSTFOCUS event, this will prevent your object from losing the focus when the mouse moves off it onto the

screen background or some inert object, so it will only lose the input focus when some other object is ready to take over the focus (this trick is used by the d_edit_proc() object).

MSG_GOTMOUSE:
MSG_LOSTMOUSE:
Sent when the mouse moves on top of or away from an object. Unlike the focus messages, these are not followed by a MSG_DRAW, so if the object is displayed differently when the mouse is on top of it, it is responsible for redrawing itself in response to these messages.

MSG_IDLE:
Sent whenever the dialog manager has nothing better to do.

MSG_RADIO:
Sent by radio button objects to deselect other buttons in the same group when they are clicked. The group number is passed in the c message parameter.

MSG_WHEEL:
Sent to the focused object whenever the mouse wheel moves. The c message parameter contains the number of clicks.

MSG_LPRESS, MSG_MPRESS, MSG_RPRESS:
Sent when the corresponding mouse button is pressed.

MSG_LRELEASE, MSG_MRELEASE, MSG_RRELEASE:
Sent when the corresponding mouse button is released.

MSG_USER:
The first free message value. Any numbers from here on (MSG_USER, MSG_USER+1, MSG_USER+2, ... MSG_USER+n) are free to use for whatever you like.

Allegro provides several standard dialog procedures. You can use these as they are to provide simple user interface objects, or you can call them from within your own dialog procedures, resulting in a kind of OOP inheritance. For instance, you could make an object which calls d_button_proc to draw itself, but handles the click message in a different way, or an object which calls d_button_proc for everything except drawing itself, so it would behave like a normal button but could look completely different.

Since the release of Allegro version 3.9.33 (CVS), some GUI objects and menus are being drawn differently unlike in previous Allegro versions. The changes are the following:

- Shadows under d_shadow_box_proc and d_button_proc are always black.
- The most important (and immediately visible) change is, that some objects are being drawn smaller. The difference is exactly one pixel in both height and width, when comparing to previous versions. The reason is, that in previous versions these objects were too large on the screen - their size was d->w+1 and d->h+1 pixels (and not d->w and d->h, as it should be). This change affects the following objects :

```
          d_box_proc,
          d_shadow_box_proc,
          d_button_proc,
          d_check_proc,
          d_radio_proc,
          d_list_proc,
          d_text_list_proc and
```

`d_textbox_proc.`

When you want to convert old dialogs to look equally when compiling with the new Allegro version, just increase the size of the mentioned objects by one pixel in both width and height fields.

- When a GUI menu item (not in a bar menu) has a child menu, there is a small arrow next to the child menu name, pointing to the right - so the user can immediately see that this menu item has a child menu - and there is no need to use such menu item names as for example "New...", to show that it has a child menu. The submenu will be drawn to the right of the parent menu, trying not to overlap it.

33.1 d_clear_proc

`int d_clear_proc(int msg, DIALOG *d, int c);`

This just clears the screen when it is drawn. Useful as the first object in a dialog.

33.2 d_box_proc

`int d_box_proc(int msg, DIALOG *d, int c);`

`int d_shadow_box_proc(int msg, DIALOG *d, int c);`

These draw boxes onto the screen, with or without a shadow.

33.3 d_bitmap_proc

`int d_bitmap_proc(int msg, DIALOG *d, int c);`

This draws a bitmap onto the screen, which should be pointed to by the dp field.

33.4 d_text_proc

`int d_text_proc(int msg, DIALOG *d, int c);`

`int d_ctext_proc(int msg, DIALOG *d, int c);`

`int d_rtext_proc(int msg, DIALOG *d, int c);`

These draw text onto the screen. The dp field should point to the string to display. d_ctext_proc() centres the string around the x coordinate, and d_rtext_proc() right aligns it. Any '&' characters in the string will be replaced with lines underneath the following character, for displaying keyboard shortcuts (as in MS Windows). To display a single ampersand, put "&&". To draw the text in something other than the default font, set the dp2 field to point to your custom font data.

33.5 d_button_proc

`int d_button_proc(int msg, DIALOG *d, int c);`

A button object (the dp field points to the text string). This object can be selected by clicking on it with the mouse or by pressing its keyboard shortcut.

If the D_EXIT flag is set, selecting it will close the dialog, otherwise it will toggle on and off. Like d_text_proc(), ampersands can be used to display the keyboard shortcut of the button.

33.6 d_check_proc

`int d_check_proc(int msg, DIALOG *d, int c);`

This is an example of how you can derive objects from other objects. Most of the functionality comes from d_button_proc(), but it displays itself as a check box. If the d1 field is non-zero, the text will be printed to the right of the check, otherwise it will be on the left.

Note: the object width should allow space for the text as well as the check box (which is square, with sides equal to the object height).

33.7 d_radio_proc

`int d_radio_proc(int msg, DIALOG *d, int c);`

A radio button object. A dialog can contain any number of radio button groups: selecting a radio button causes other buttons within the same group to be deselected. The dp field points to the text string, d1 specifies the group number, and d2 is the button style (0=circle, 1=square).

33.8 d_icon_proc

`int d_icon_proc(int msg, DIALOG *d, int c);`

A bitmap button. The fg color is used for the dotted line showing focus, and the bg color for the shadow used to fill in the top and left sides of the button when "pressed". d1 is the "push depth", ie. the number of pixels the icon will be shifted to the right and down when selected (default 2) if there is no "selected" image. d2 is the distance by which the dotted line showing focus is indented (default 2). dp points to a bitmap for the icon, while dp2 and dp3 are the selected and disabled images respectively (optional, may be NULL).

33.9 d_keyboard_proc

`int d_keyboard_proc(int msg, DIALOG *d, int c);`

This is an invisible object for implementing keyboard shortcuts. You can put an ASCII code in the key field of the dialog object (a character such as 'a' to respond to a simple keypress, or a number 1-26 to respond to a control key a-z), or you can put a keyboard scancode in the d1 and/or d2 fields. When one of these keys is pressed, the object will call the function pointed to by dp. This should return an int, which will be passed back to the dialog manager, so it can return D_O_K, D_REDRAW, D_CLOSE, etc.

33.10 d_edit_proc

`int d_edit_proc(int msg, DIALOG *d, int c);`

An editable text object (the dp field points to the string). When it has the input focus (obtained by clicking on it with the mouse), text can be typed into this object. The d1 field specifies the maximum number of characters that it will accept, and d2 is the text cursor position within the string.

33.11 d_list_proc

`int d_list_proc(int msg, DIALOG *d, int c);`

A list box object. This will allow the user to scroll through a list of items and to select one by clicking or with the arrow keys. If the D_EXIT flag is set, double clicking on a list item will close the dialog. The index of the selected item is held in the d1 field, and d2 is used to store how far it has scrolled through the list. The dp field points to a function which will be called to obtain information about the contents of the list. This should follow the form:

```
     char *foobar(int index, int *list_size);
```

If index is zero or positive, the function should return a pointer to the string which is to be displayed at position index in the list. If index is negative, it should return NULL and list_size should be set to the number of items in the list.

To create a multiple selection listbox, set the dp2 field to an array of byte flags indicating the selection state of each list item (non-zero for selected entries). This table must be at least as big as the number of objects in the list!

33.12 d_text_list_proc

`int d_text_list_proc(int msg, DIALOG *d, int c);`

Like d_list_proc, but allows the user to type in the first few characters of a listbox entry in order to select it. Uses dp3 internally, so you mustn't store anything important there yourself.

33.13 d_textbox_proc

`int d_textbox_proc(int msg, DIALOG *d, int c);`

A text box object. The dp field points to the text which is to be displayed in the box. If the text is long, there will be a vertical scrollbar on the right hand side of the object which can be used to scroll through the text. The default is to print the text with word wrapping, but if the D_SELECTED flag is set, the text will be printed with character wrapping. The d1 field is used internally to store the number of lines of text, and d2 is used to store how far it has scrolled through the text.

33.14 d_slider_proc

`int d_slider_proc(int msg, DIALOG *d, int c);`

A slider control object. This object holds a value in d2, in the range from 0 to d1. It will display as a vertical slider if h is greater than or equal to w, otherwise it will display as a horizontal slider. The dp field can contain an optional bitmap to use for the slider handle, and dp2 can contain an optional callback function, which is called each time d2 changes. The callback function should have the following prototype:

```
int function(void *dp3, int d2);
```

The d_slider_proc object will return the value of the callback function.

33.15 d_menu_proc

`int d_menu_proc(int msg, DIALOG *d, int c);`

This object is a menu bar which will drop down child menus when it is clicked or if an alt+key corresponding to one of the shortcuts in the menu is pressed. It ignores a lot of the fields in the dialog structure, in particular the color is taken from the gui_*_color variables, and the width and height are calculated automatically (the w and h fields from the DIALOG are only used as a minimum size.) The dp field points to an array of menu structures: see do_menu() for more information. The top level menu will be displayed as a horizontal bar, but when child menus drop down from it they will be in the normal vertical format used by do_menu(). When a menu item is selected, the return value from the menu callback function is passed back to the dialog manager, so your callbacks should return D_O_K, D_REDRAW, or D_CLOSE.

See also:

See Section 33.39 [gui menus], page 223.

See Section 33.41 [active_menu], page 224.

See Section 33.42 [gui_menu_draw_menu], page 224.

33.16 d_yield_proc

`int d_yield_proc(int msg, DIALOG *d, int c);`

An invisible helper object that yields timeslices for the scheduler (if the system supports it) when the gui has nothing to do but waiting for user actions. You should put one instance of this object in each dialog array because it may be needed on systems with an unusual scheduling algorithm (for instance QNX) in order to make the GUI fully responsive.

33.17 gui_mouse_focus

`extern int gui_mouse_focus;`

If set, the input focus follows the mouse pointer around the dialog, otherwise a click is required to move it.

33.18 gui_fg_color

`extern int gui_fg_color;`

`extern int gui_bg_color;`

The foreground and background colors for the standard dialogs (alerts, menus, and the file selector). They default to 255 and 0.

See also:

See Section 33.19 [gui_mg_color], page 218.

See Section 33.27 [set_dialog_color], page 219.

33.19 gui_mg_color

`extern int gui_mg_color;`

The color used for displaying greyed-out dialog objects (with the D_DISABLED flag set). Defaults to 8.

See also:

See Section 33.18 [gui_fg_color], page 217.

See Section 33.27 [set_dialog_color], page 219.

33.20 gui_font_baseline

`extern int gui_font_baseline;`

If set to a non-zero value, adjusts the keyboard shortcut underscores to account for the height of the descenders in your font.

33.21 gui_mouse_x

`extern int (*gui_mouse_x)();`

`extern int (*gui_mouse_y)();`

`extern int (*gui_mouse_z)();`

`extern int (*gui_mouse_b)();`

Hook functions, used by the GUI routines whenever they need to access the mouse state. By default these just return copies of the mouse_x, mouse_y, mouse_z, and mouse_b variables, but they could be used to offset or scale the mouse position, or read input from a different source entirely.

33.22 gui font

You can change the global 'font' pointer to make the GUI objects use something other than the standard 8x8 font. The standard dialog procedures, menus, and alert boxes, will work with fonts of any size, but the gfx_mode_select() dialog will look wrong with anything other than 8x8 fonts.

33.23 gui_textout

`int gui_textout(BITMAP *bmp, const char *s, int x, y, color, centre);`

Helper function for use by the GUI routines. Draws a text string onto the screen, interpreting the '&' character as an underbar for displaying keyboard shortcuts. Returns the width of the output string in pixels.

See also:

See Section 33.24 [gui_strlen], page 219.

33.24 gui_strlen

```
int gui_strlen(const char *s);
```

Helper function for use by the GUI routines. Returns the length of a string in pixels, ignoring '&' characters.

See also:

See Section 33.23 [gui_textout], page 218.

33.25 position_dialog

```
void position_dialog(DIALOG *dialog, int x, int y);
```

Moves an array of dialog objects to the specified screen position (specified as the top left corner of the dialog).

See also:

See Section 33.26 [centre_dialog], page 219.

33.26 centre_dialog

```
void centre_dialog(DIALOG *dialog);
```

Moves an array of dialog objects so that it is centered in the screen.

See also:

See Section 33.25 [position_dialog], page 219.

See Section 33.27 [set_dialog_color], page 219.

33.27 set_dialog_color

```
void set_dialog_color(DIALOG *dialog, int fg, int bg);
```

Sets the foreground and background colors of an array of dialog objects.

See also:

See Section 33.18 [gui_fg_color], page 217.

See Section 33.19 [gui_mg_color], page 218.

See Section 33.26 [centre_dialog], page 219.

33.28 find_dialog_focus

`int find_dialog_focus(DIALOG *dialog);`

Searches the dialog for the object which has the input focus, returning an index or -1 if the focus is not set. This is useful if you are calling do_dialog() several times in a row and want to leave the focus in the same place it was when the dialog was last displayed, as you can call do_dialog(dlg, find_dialog_focus(dlg));

See also:

See Section 33.33 [do_dialog], page 221.

See Section 33.35 [init_dialog], page 221.

See Section 33.29 [offer_focus], page 220.

33.29 offer_focus

`int offer_focus(DIALOG *d, int obj, int *focus_obj, int force);`

Offers the input focus to a particular object. Normally the function sends the MSG_WANTFOCUS message to query whether the object is willing to accept the focus. However, passing any non zero value as force argument instructs the function to authoritatively set the focus to the object.

See also:

See Section 33.28 [find_dialog_focus], page 220.

33.30 object_message

`int object_message(DIALOG *dialog, int msg, int c);`

Sends a message to an object and returns the answer it has generated. Remember that the first parameter is the dialog object (not a whole array) that you wish to send the message to. For example, to make the second object in a dialog draw itself, you might write:

```
object_message(&dialog[1], MSG_DRAW, 0);
```

See also:

See Section 33.31 [dialog_message], page 220.

33.31 dialog_message

`int dialog_message(DIALOG *dialog, int msg, int c, int *obj);`

Sends a message to all the objects in an array. If any of the dialog procedures return values other than D_O_K, it returns the value and sets obj to the index of the object which produced it.

See also:

See Section 33.30 [object_message], page 220.

See Section 33.32 [broadcast_dialog_message], page 221.

33.32 broadcast_dialog_message

`int broadcast_dialog_message(int msg, int c);`

Broadcasts a message to all the objects in the active dialog. If any of the dialog procedures return values other than D_O_K, it returns that value.

See also:

See Section 33.31 [dialog_message], page 220.

See Section 33.38 [active_dialog], page 222.

33.33 do_dialog

`int do_dialog(DIALOG *dialog, int focus_obj);`

The basic dialog manager function. This displays a dialog (an array of dialog objects, terminated by one with a NULL dialog procedure), and sets the input focus to the focus_obj (-1 if you don't want anything to have the focus). It interprets user input and dispatches messages as they are required, until one of the dialog procedures tells it to close the dialog, at which point it returns the index of the object that caused it to exit.

See also:

See Section 33.34 [popup_dialog], page 221.

See Section 33.35 [init_dialog], page 221.

See Section 33.26 [centre_dialog], page 219.

See Section 33.27 [set_dialog_color], page 219.

See Section 33.28 [find_dialog_focus], page 220.

33.34 popup_dialog

`int popup_dialog(DIALOG *dialog, int focus_obj);`

Like do_dialog(), but it stores the data on the screen before drawing the dialog and restores it when the dialog is closed. The screen area to be stored is calculated from the dimensions of the first object in the dialog, so all the other objects should lie within this one.

See also:

See Section 33.33 [do_dialog], page 221.

33.35 init_dialog

`DIALOG_PLAYER *init_dialog(DIALOG *dialog, int focus_obj);`

This function provides lower level access to the same functionality as do_dialog(), but allows you to combine a dialog box with your own program control structures. It initialises a dialog, returning a pointer to a player object that can be used with update_dialog() and shutdown_dialog(). With these functions, you could implement your own version of do_dialog() with the lines:

```
DIALOG_PLAYER *player = init_dialog(dialog, focus_obj);

while (update_dialog(player))
   ;

return shutdown_dialog(player);
```

See also:

See Section 33.36 [update_dialog], page 222.

See Section 33.37 [shutdown_dialog], page 222.

See Section 33.33 [do_dialog], page 221.

33.36 update_dialog

`int update_dialog(DIALOG_PLAYER *player);`

Updates the status of a dialog object returned by init_dialog(). Returns TRUE if the dialog is still active, or FALSE if it has terminated. Upon a return value of FALSE, it is up to you whether to call shutdown_dialog() or to continue execution. The object that requested the exit can be determined from the player->obj field.

See also:

See Section 33.35 [init_dialog], page 221.

33.37 shutdown_dialog

`int shutdown_dialog(DIALOG_PLAYER *player);`

Destroys a dialog player object returned by init_dialog(), returning the object that caused it to exit (this is the same as the return value from do_dialog()).

See also:

See Section 33.35 [init_dialog], page 221.

33.38 active_dialog

```
extern DIALOG *active_dialog;
```

Global pointer to the most recent activated dialog. This may be useful if an object needs to iterate through a list of all its siblings.

See also:

See Section 33.33 [do_dialog], page 221.

See Section 33.35 [init_dialog], page 221.

See Section 33.32 [broadcast_dialog_message], page 221.

33.39 gui menus

Popup or pulldown menus are created as an array of the structures:

```
     typedef struct MENU
     {
        char *text;                    - the text to display for the menu item
        int (*proc)(void);             - called when the menu item is clicked
        struct MENU *child;            - nested child menu
        int flags;                     - disabled or checked state
        void *dp;                      - pointer to any data you need
     } MENU;
```

Each menu item contains a text string. This can use the '&' character to indicate keyboard shortcuts, or can be an zero-length string to display the item as a non-selectable splitter bar. If the string contains a "\t" tab character, any text after this will be right-justified, eg. for displaying keyboard shortcut information. The proc pointer is a function which will be called when the menu item is selected, and child points to another menu, allowing you to create nested menus. Both proc and child may be NULL. The proc function returns an integer which is ignored if the menu was brought up by calling do_menu(), but which is passed back to the dialog manager if it was created by a d_menu_proc() object. The array of menu items is terminated by an entry with a NULL text pointer.

Menu items can be disabled (greyed-out) by setting the D_DISABLED bit in the flags field, and a check mark can be displayed next to them by setting the D_SELECTED bit. With the default alignment and font this will usually overlap the menu text, so if you are going to use checked menu items it would be a good idea to prefix all your options with a space or two, to ensure there is room for the check.

See also:

See Section 33.40 [do_menu], page 223.

See Section 33.15 [d_menu_proc], page 217.

See Section 33.42 [gui_menu_draw_menu], page 224.

33.40 do_menu

```
int do_menu(MENU *menu, int x, int y)
```

Displays and animates a popup menu at the specified screen coordinates (these will be adjusted if the menu does not entirely fit on the screen). Returns the index of the menu item that was selected, or -1 if the menu was cancelled. Note that the return value cannot indicate selection from child menus, so you will have to use the callback functions if you want multi-level menus.

See also:

See Section 33.39 [gui menus], page 223.

See Section 33.15 [d_menu_proc], page 217.

See Section 33.41 [active_menu], page 224.

See Section 33.42 [gui_menu_draw_menu], page 224.

33.41 active_menu

```
extern MENU *active_menu;
```

When a menu callback procedure is triggered, this will be set to the menu item that was selected, so your routine can determine where it was called from.

See also:

See Section 33.39 [gui menus], page 223.

33.42 gui_menu_draw_menu

```
extern void (*gui_menu_draw_menu)(int x, int y, int w, int h);
extern void (*gui_menu_draw_menu_item)(MENU *m, int x, int y, int w, int h,
int bar, int sel);
```

If set, these functions will be called whenever a menu needs to be drawn, so you can change how menus look.

gui_menu_draw_menu() is passed the position and size of the menu. It should draw the background of the menu onto screen.

gui_menu_draw_menu_item() is called once for each menu item that is to be drawn. bar will be set if the item is part of a top-level horizontal menu bar, and sel will be set if the menu item is selected. It should also draw onto screen.

See also:

See Section 33.39 [gui menus], page 223.

33.43 alert

```
int alert(const char *s1, *s2, *s3, const char *b1, *b2, int c1, c2);
```

Displays a popup alert box, containing three lines of text (s1-s3), and with either one or two buttons. The text for these buttons is passed in b1 and b2 (b2 may be NULL), and the keyboard shortcuts in c1 and c2. Returns 1 or 2 depending on which button was clicked. If the alert is dismissed by pressing ESC when ESC is not one of the keyboard shortcuts, it treats it as a click on the second button (this is consistent with the common "Ok", "Cancel" alert).

See also:

See Section 33.44 [alert3], page 225.

See Section 33.18 [gui_fg_color], page 217.

33.44 alert3

`int alert3(const char *s1, *s2, *s3, const char *b1, *b2, *b3, int c1, c2, c3);`

Like alert(), but with three buttons. Returns 1, 2, or 3.

See also:

See Section 33.43 [alert], page 224.

See Section 33.18 [gui_fg_color], page 217.

33.45 file_select

`int file_select(const char *message, char *path, const char *ext);`

Deprecated. Use file_select_ex() instead, passing the two constants OLD_FILESEL_WIDTH and OLD_FILESEL_HEIGHT if you want the file selector to be displayed with the dimensions of the old file selector.

See also:

See Section 33.46 [file_select_ex], page 225.

See Section 33.18 [gui_fg_color], page 217.

33.46 file_select_ex

`int file_select_ex(const char *message, char *path, const char *ext, int size, int w, int h);`

Displays the Allegro file selector, with the message as caption. The path parameter contains the initial filename to display (this can be used to set the starting directory, or to provide a default filename for a save-as operation). The user selection is returned by altering the path buffer, whose maximum capacity in bytes is specified by the size parameter. Note that it should have room for at least 80 characters (not bytes), so you should reserve 6x that amount, just to be sure. The list of files is filtered according to the file extensions in the ext parameter. Passing NULL includes all files; "PCX;BMP" includes only files with .PCX or .BMP extensions. If you wish to control files by their attributes, one of the fields in the extension list can begin with a slash, followed by a set of attribute characters. Any attributes written on their own, or with a + before them, indicate to include only files which have that attribute set. Any attributes with a '-' before them indicate to leave out any files with that attribute. The flag characters are 'r' (read-only), 'h' (hidden), 's' (system), 'd' (directory), and 'a' (archive). For example, an extension string of "PCX;BMP;/+r-d" will display only PCX or BMP files that are read-only, and no directories. The file selector is stretched to the width and height specified in the w and h parameters, and to the size of the standard Allegro font. If either the width or height argument is set to zero, it is stretched to the corresponding screen dimension. This function returns zero if it was closed with the Cancel button or non-zero if it was OK'd.

See also:

See Section 33.18 [gui_fg_color], page 217.

33.47 gfx_mode_select

`int gfx_mode_select(int *card, int *w, int *h);`

Displays the Allegro graphics mode selection dialog, which allows the user to select a screen mode and graphics card. Stores the selection in the three variables, and returns zero if it was closed with the Cancel button or non-zero if it was OK'd.

See also:

See Section 33.48 [gfx_mode_select_ex], page 226.

See Section 8.6 [set_gfx_mode], page 70.

See Section 33.18 [gui_fg_color], page 217.

33.48 gfx_mode_select_ex

`int gfx_mode_select_ex(int *card, int *w, int *h, int *color_depth);`

Extended version of the graphics mode selection dialog, which allows the user to select the color depth as well as the resolution and hardware driver. This version of the function reads the initial values from the parameters when it activates, so you can specify the default values.

See also:

See Section 33.47 [gfx_mode_select], page 226.

See Section 8.1 [set_color_depth], page 68.

See Section 8.6 [set_gfx_mode], page 70.

See Section 33.18 [gui_fg_color], page 217.

33.49 gui_shadow_box_proc

```
extern int (*gui_shadow_box_proc)(int msg, struct DIALOG *d, int c);
extern int (*gui_ctext_proc)(int msg, struct DIALOG *d, int c);
extern int (*gui_button_proc)(int msg, struct DIALOG *d, int c);
extern int (*gui_edit_proc)(int msg, struct DIALOG *d, int c);
extern int (*gui_list_proc)(int msg, struct DIALOG *d, int c);
extern int (*gui_text_list_proc)(int msg, struct DIALOG *d, int c);
```

If set, these functions will be used by the standard Allegro dialogs. This allows you to customise the look and feel, much like gui_fg_color and gui_bg_color, but much more flexiblely.

See also:
See Section 33.43 [alert], page 224.
See Section 33.44 [alert3], page 225.
See Section 33.45 [file_select], page 225.
See Section 33.47 [gfx_mode_select], page 226.
See Section 33.18 [gui_fg_color], page 217.

34 DOS specifics

34.1 JOY_TYPE_*/DOS

`Drivers JOY_TYPE_*/DOS`

The DOS library supports the following type parameters for the install_joystick() function:

- JOY_TYPE_AUTODETECT
  Attempts to autodetect your joystick hardware. It isn't possible to reliably distinguish between all the possible input setups, so this routine can only ever choose the standard joystick, Sidewider, GamePad Pro, or GrIP drivers, but it will use information from the configuration file if one is available (this can be created using the setup utility or by calling the save_joystick_data() function), so you can always use JOY_TYPE_AUTODETECT in your code and then select the exact hardware type from the setup program.
- JOY_TYPE_NONE
  Dummy driver for machines without any joystick.
- JOY_TYPE_STANDARD
  A normal two button stick.
- JOY_TYPE_2PADS
  Dual joystick mode (two sticks, each with two buttons).
- JOY_TYPE_4BUTTON
  Enable the extra buttons on a 4-button joystick.

- JOY_TYPE_6BUTTON
  Enable the extra buttons on a 6-button joystick.
- JOY_TYPE_8BUTTON
  Enable the extra buttons on an 8-button joystick.
- JOY_TYPE_FSPRO
  CH Flightstick Pro or compatible stick, which provides four buttons, an analogue throttle control, and a 4-direction coolie hat.
- JOY_TYPE_WINGEX
  A Logitech Wingman Extreme, which should also work with any Thrustmaster Mk.I compatible joystick. It provides support for four buttons and a coolie hat. This also works with the Wingman Warrior, if you plug in the 15 pin plug (remember to unplug the 9-pin plug!) and set the tiny switch in front to the "H" position (you will not be able to use the throttle or the spinner though).
- JOY_TYPE_SIDEWINDER
  The Microsoft Sidewinder digital pad (supports up to four controllers, each with ten buttons and a digital direction control).
- JOY_TYPE_SIDEWINDER_AG
  An alternative driver to JOY_TYPE_SIDEWINDER. Try this if your Sidewinder isn't recognized with JOY_TYPE_SIDEWINDER.
- JOY_TYPE_GAMEPAD_PRO
  The Gravis GamePad Pro (supports up to two controllers, each with ten buttons and a digital direction control).
- JOY_TYPE_GRIP
  Gravis GrIP driver, using the grip.gll driver file.
- JOY_TYPE_GRIP4
  Version of the Gravis GrIP driver that is constrained to only move along the four main axis.
- JOY_TYPE_SNESPAD_LPT1
  JOY_TYPE_SNESPAD_LPT2
  JOY_TYPE_SNESPAD_LPT3
  SNES joypads connected to LPT1, LPT2, and LPT3 respectively.
- JOY_TYPE_PSXPAD_LPT1
  JOY_TYPE_PSXPAD_LPT2
  JOY_TYPE_PSXPAD_LPT3
  PSX joypads connected to LPT1, LPT2, and LPT3 respectively. See http://www.ziplabel.com/dpadpro/index.html for information about the parallel cable required. The driver automagically detects which types of PSX pads are connected out of digital, analog (red or green mode), NegCon, multi taps, Namco light guns, Jogcons (force feedback steering wheel) and the mouse. If the controller isn't recognised it is treated as an analog controller, meaning the driver should work with just about anything. You can connect controllers in any way you see fit, but only the first 8 will be used.

The Sony Dual Shock or Namco Jogcon will reset themselves (to digital mode) after not being polled for 5 seconds. This is normal, the same thing happens on a Playstation, it's designed to stop any vibration in case the host machine crashes. Other mode switching controllers may have similar quirks. However, if this happens to a Jogcon controller the mode button is disabled. To reenable the mode button on the Jogcon you need to hold down the Start and Select buttons at the same time.

The G-con45 needs to be connected to (and pointed at) a TV type monitor connected to your computer. The composite video out on my video card works fine for this (a Hercules Stingray 128/3D 8Mb). The TV video modes in Mame should work too.

- JOY_TYPE_N64PAD_LPT1
  JOY_TYPE_N64PAD_LPT2
  JOY_TYPE_N64PAD_LPT3
  N64 joypads connected to LPT1, LPT2, and LPT3 respectively. See http://www.st-hans.de/N64.htm for information about the necessary hardware adaptor. It supports up to four controllers on a single parallel port. There is no need to calibrate the analog stick, as this is done by the controller itself when powered up. This means that the stick has to be centred when the controller is initialised. One possible issue people may have with this driver is that it is physically impossible to move the analog stick fully diagonal, but I can't see this causing any major problems. This is because of the shape of the rim that the analog stick rests against. Like the Gravis Game Pad Pro, this driver briefly needs to disable hardware interrupts while polling. This causes a noticable performance hit on my machine in both drivers, but there is no way around it. At a (very) rough guess I'd say it slows down Mame 5% - 10%.
- JOY_TYPE_DB9_LPT1
  JOY_TYPE_DB9_LPT2
  JOY_TYPE_DB9_LPT3
  A pair of two-button joysticks connected to LPT1, LPT2, and LPT3 respectively. Port 1 is compatible with Linux joy-db9 driver (multisystem 2-button), and port 2 is compatible with Atari interface for DirectPad Pro. See the source file (src/dos/multijoy.c) for pinout information.
- JOY_TYPE_TURBOGRAFIX_LPT1
  JOY_TYPE_TURBOGRAFIX_LPT2
  JOY_TYPE_TURBOGRAFIX_LPT3
  These drivers support up to 7 joysticks, each one with up to 5 buttons, connected to LPT1, LPT2, and LPT3 respectively. They use the TurboGraFX interface by Steffen Schwenke: see http://www.burg-halle.de/~schwenke/parport.html for details on how to build this.
- JOY_TYPE_WINGWARRIOR
  A Wingman Warrior joystick.
- JOY_TYPE_IFSEGA_ISA
  JOY_TYPE_IFSEGA_PCI

JOY_TYPE_IFSEGA_PCI_FAST
Drivers for the IF-SEGA joystick interface cards by the IO-DATA company (these come in PCI, PCI2, and ISA variants).

See also:

See Section 7.1 [install_joystick], page 63.

34.2 GFX_*/DOS

`Drivers GFX_*/DOS`

The DOS library supports the following card parameters for the set_gfx_mode() function:

- GFX_TEXT
  Return to text mode.
- GFX_AUTODETECT
  Let Allegro pick an appropriate graphics driver.
- GFX_AUTODETECT_FULLSCREEN
  Autodetects a graphics driver, but will only use fullscreen drivers, failing if these are not available on current platform.
- GFX_AUTODETECT_WINDOWED
  Same as above, but uses only windowed drivers. This will always fail under DOS.
- GFX_SAFE
  Special driver for when you want to reliably set a graphics mode and don't really care what resolution or color depth you get. See the set_gfx_mode() documentation for details.
- GFX_VGA
  The standard 256 color VGA mode 13h, using the GFX_VGA driver. This is normally sized 320x200, which will work on any VGA but doesn't support large virtual screens and hardware scrolling. Allegro also provides some tweaked variants of the mode which are able to scroll, sized 320x100 (with a 200 pixel high virtual screen), 160x120 (with a 409 pixel high virtual screen), 256x256 (no scrolling), and 80x80 (with a 819 pixel high virtual screen).
- GFX_MODEX
  Mode-X will work on any VGA card, and provides a range of different 256 color tweaked resolutions.
  - Stable mode-X resolutions:
    - Square aspect ratio: 320x240
    - Skewed aspect ratio: 256x224, 256x240, 320x200, 320x400, 320x480, 320x600, 360x200, 360x240, 360x360, 360x400, 360x480

    These have worked on every card/monitor that I've tested.
  - Unstable mode-X resolutions:

    - Square aspect ratio: 360x270, 376x282, 400x300
    - Skewed aspect ratio: 256x200, 256x256, 320x350, 360x600, 376x308, 376x564, 400x150, 400x600

  These only work on some monitors. They were fine on my old machine, but don't get on very well with my new monitor. If you are worried about the possibility of damaging your monitor by using these modes, don't be. Of course I'm not providing any warranty with any of this, and if your hardware does blow up that is tough, but I don't think this sort of tweaking can do any damage. From the documentation of Robert Schmidt's TWEAK program:

  "Some time ago, putting illegal or unsupported values or combinations of such into the video card registers might prove hazardous to both your monitor and your health. I have *never* claimed that bad things can't happen if you use TWEAK, although I'm pretty sure it never will. I've never heard of any damage arising from trying out TWEAK, or from general VGA tweaking in any case."

  Most of the mode-X drawing functions are slower than in mode 13h, due to the complexity of the planar bitmap organisation, but solid area fills and plane-aligned blits from one part of video memory to another can be significantly faster, particularly on older hardware. Mode-X can address the full 256k of VGA RAM, so hardware scrolling and page flipping are possible, and it is possible to split the screen in order to scroll the top part of the display but have a static status indicator at the bottom.

- GFX_VESA1
  Use the VESA 1.x driver.
- GFX_VESA2B
  Use the VBE 2.0 banked mode driver.
- GFX_VESA2L
  Use the VBE 2.0 linear framebuffer driver.
- GFX_VESA3
  Use the VBE 3.0 driver. This is the only VESA driver that supports the request_refresh_rate() function.

  The standard VESA modes are 640x480, 800x600, and 1024x768. These ought to work with any SVGA card: if they don't, get a copy of the SciTech Display Doctor and see if that fixes it. What color depths are available will depend on your hardware. Most cards support both 15 and 16 bit resolutions, but if at all possible I would advise you to support both (it's not hard...) in case one is not available. Some cards provide both 24 and 32 bit truecolor, in which case it is a choice between 24 (saves memory) or 32 (faster), but many older cards have no 32 bit mode and some newer ones don't support 24 bit resolutions. Use the vesainfo test program to see what modes your VESA driver provides.

  Many cards also support 640x400, 1280x1024, and 1600x1200, but these aren't available on everything, for example the S3 chipset has no 640x400 mode. Other weird resolution may be possible, eg. some Tseng boards can do 640x350, and the Avance Logic has a 512x512 mode.

The SciTech Display Doctor provides several scrollable low resolution modes in a range of different color depths (320x200, 320x240, 320x400, 320x480, 360x200, 360x240, 360x400, and 360x480 all work on my ET4000 with 8, 15, or 16 bits per pixel). These are lovely, allowing scrolling and page flipping without the complexity of the mode-X planar setup, but unfortunately they aren't standard so you will need Display Doctor in order to use them.

- GFX_VBEAF
  VBE/AF is a superset of the VBE 2.0 standard, which provides an API for accessing hardware accelerator features. VBE/AF drivers are currently only available from the FreeBE/AF project or as part of the SciTech Display Doctor package, but they can give dramatic speed improvements when used with suitable hardware. For a detailed discussion of hardware acceleration issues, refer to the documentation for the gfx_capabilities flag.

  You can use the afinfo test program to check whether you have a VBE/AF driver, and to see what resolutions it supports.

  The SciTech VBE/AF drivers require nearptr access to be enabled, so any stray pointers are likely to crash your machine while their drivers are in use. This means it may be a good idea to use VESA while debugging your program, and only switch to VBE/AF once the code is working correctly. The FreeBE/AF drivers do not have this problem.
- GFX_XTENDED
  An unchained 640x400 mode, as described by Mark Feldman in the PCGPE. This uses VESA to select an SVGA mode (so it will only work on cards supporting the VESA 640x400 resolution), and then unchains the VGA hardware as for mode-X. This allows the entire screen to be addressed without the need for bank switching, but hardware scrolling and page flipping are not possible. This driver will never be autodetected (the normal VESA 640x400 mode will be chosen instead), so if you want to use it you will have to explicitly pass GFX_XTENDED to set_gfx_mode().

See also:

See Section 8.6 [set_gfx_mode], page 70.

34.3 DIGI_*/DOS

`Drivers DIGI_*/DOS`

The DOS sound functions support the following digital soundcards:

```
DIGI_AUTODETECT      - let Allegro pick a digital sound driver
DIGI_NONE            - no digital sound
DIGI_SB              - Sound Blaster (autodetect type)
DIGI_SB10            - SB 1.0 (8 bit mono single shot dma)
```

```
          DIGI_SB15            - SB 1.5 (8 bit mono single shot dma)
          DIGI_SB20            - SB 2.0 (8 bit mono auto-initialised
                                 dma)
          DIGI_SBPRO           - SB Pro (8 bit stereo)
          DIGI_SB16            - SB16 (16 bit stereo)
          DIGI_AUDIODRIVE      - ESS AudioDrive
          DIGI_SOUNDSCAPE      - Ensoniq Soundscape
          DIGI_WINSOUNDSYS     - Windows Sound System
```

See also:

See Section 23.1 [detect_digi_driver], page 157.

See Section 23.5 [install_sound], page 160.

See Section 27.1 [install_sound_input], page 177.

34.4 MIDI_*/DOS

`Drivers MIDI_*/DOS`

The DOS sound functions support the following MIDI soundcards:

```
          MIDI_AUTODETECT      - let Allegro pick a MIDI sound driver
          MIDI_NONE            - no MIDI sound
          MIDI_ADLIB           - Adlib or SB FM synth (autodetect type)
          MIDI_OPL2            - OPL2 synth (mono, used in Adlib and SB)
          MIDI_2XOPL2          - dual OPL2 synths (stereo, used in
                                 SB Pro-I)
          MIDI_OPL3            - OPL3 synth (stereo, SB Pro-II
                                 and above)
          MIDI_SB_OUT          - SB MIDI interface
          MIDI_MPU             - MPU-401 MIDI interface
          MIDI_DIGMID          - sample-based software wavetable player
          MIDI_AWE32           - AWE32 (EMU8000 chip)
```

See also:

See Section 23.2 [detect_midi_driver], page 158.

See Section 23.5 [install_sound], page 160.

See Section 27.1 [install_sound_input], page 177.

34.5 split_modex_screen

`void split_modex_screen(int line);`

This function is only available in mode-X. It splits the VGA display into two parts at the specified line. The top half of the screen can be scrolled to any part of video memory with the scroll_screen() function, but the part below the specified line number will remain fixed and will display from position (0, 0) of the screen bitmap. After splitting the screen you will generally want to scroll so that the top part of the display starts lower down in video memory, and then create two sub-bitmaps to access the two sections (see examples/exscroll.c for a demonstration of how this could be done). To disable the split, call split_modex_screen(0).

See also:

See Section 8.6 [set_gfx_mode], page 70.

See Section 8.13 [scroll_screen], page 77.

34.6 i_love_bill

```
extern int i_love_bill;
```

When running in clean DOS mode, the timer handler dynamically reprograms the clock chip to generate interrupts at exactly the right times, which gives an extremely high accuracy. Unfortunately, this constant speed adjustment doesn't work under most multitasking systems (notably Windows), so there is an alternative mode that just locks the hardware timer interrupt to a speed of 200 ticks per second. This reduces the accuracy of the timer (for instance, rest() will round the delay time to the nearest 5 milliseconds), and prevents the vertical retrace simulator from working, but on the plus side, it makes Allegro programs work under Windows. This flag is set by allegro_init() if it detects the presence of a multitasking OS, and enables the fixed rate timer mode.

See also:
See Section 5.1 [install_timer], page 48.
See Section 1.2 [allegro_init], page 1.
See Section 1.7 [os_type], page 3.

35 Windows specifics

A Windows program that uses the Allegro library is only required to include one or more header files from the include/allegro tree, or allegro.h; however, if it also needs to directly call non portable Win32 API functions, it must include the Windows-specific header file winalleg.h after the Allegro headers, and before any Win32 API header file. By default winalleg.h includes the main Win32 C API header file windows.h. If instead you want to use the C++ interface to the Win32 API (a.k.a. the Microsoft Foundation Classes), define the preprocessor symbol ALLEGRO_AND_MFC before including any Allegro header so that afxwin.h will be included. Note that, in this latter case, the Allegro debugging macros ASSERT() and TRACE() are renamed AL_ASSERT() and AL_TRACE() respectively.

Windows GUI applications start with a WinMain() entry point, rather than the standard main() entry point. Allegro is configured to build GUI applications by default and to do some magic in order to make a regular main() work with them, but you have to help it out a bit by writing END_OF_MAIN() right after your main() function. If you don't want to do that, you can just include winalleg.h and write a WinMain() function. Note that this magic may bring about conflicts with a few programs using direct calls to Win32 API functions;

for these programs, the regular WinMain() is required and the magic must be disabled by defining the preprocessor symbol ALLEGRO_NO_MAGIC_MAIN before including Allegro headers.

If you want to build a console application using Allegro, you have to define the preprocessor symbol USE_CONSOLE before including Allegro headers; it will instruct the library to use console features and also to disable the special processing of the main() function described above.

When creating the main window, Allegro searches the executable for an ICON resource named "allegro_icon". If it is present, Allegro automatically loads it and uses it as its application icon; otherwise, Allegro uses the default IDI_APPLICATION icon. See the manual of your compiler for a method to create an ICON resource, or use the wfixicon utility from the tools/win directory.

DirectX requires that system and video bitmaps (including the screen) be locked before you can draw onto them. This will be done automatically, but you can usually get much better performance by doing it yourself: see the acquire_bitmap() function for details.

Due to a major oversight in the design of DirectX, there is no way to preserve the contents of video memory when the user switches away from your program. You need to be prepared for the fact that your screen contents, and the contents of any video memory bitmaps, may be destroyed at any point. You can use the set_display_switch_callback() function to find out when this happens.

On the Windows platform, the only return values for the desktop_color_depth() function are 8, 16, 24 and 32. This means that 15-bit and 16-bit desktops cannot be differentiated and are both reported as 16-bit desktops. See below for the consequences for windowed and overlay DirectX drivers.

35.1 GFX_*/Windows

`Drivers GFX_*/Windows`

The Windows library supports the following card parameters for the set_gfx_mode() function:

- GFX_TEXT
  This closes any graphic mode previously opened with set_gfx_mode.
- GFX_AUTODETECT
  Let Allegro pick an appropriate graphics driver.
- GFX_AUTODETECT_FULLSCREEN
  Autodetects a graphics driver, but will only use fullscreen drivers, failing if these are not available on current platform.
- GFX_AUTODETECT_WINDOWED
  Same as above, but uses only windowed drivers.
- GFX_SAFE
  Special driver for when you want to reliably set a graphics mode and don't really care what resolution or color depth you get. See the set_gfx_mode() documentation for details.
- GFX_DIRECTX
  Alias for GFX_DIRECTX_ACCEL.

- GFX_DIRECTX_ACCEL
  The regular fullscreen DirectX driver, running with hardware acceleration enabled.
- GFX_DIRECTX_SOFT
  DirectX fullscreen driver that only uses software drawing, rather than any hardware accelerated features.
- GFX_DIRECTX_SAFE
  Simplified fullscreen DirectX driver that doesn't support any hardware acceleration, video or system bitmaps, etc.
- GFX_DIRECTX_WIN
  The regular windowed DirectX driver, running in color conversion mode when the color depth doesn't match that of the Windows desktop. Color conversion is much slower than direct drawing and is not supported between 15-bit and 16-bit color depths. This limitation is needed to work around that of desktop_color_depth() (see above) and allows to select the direct drawing mode in a reliable way on desktops reported as 16-bit:

  ```
  if (desktop_color_depth() == 16) {
     set_color_depth(16);
     if (set_gfx_mode(GFX_DIRECTX_WIN, 640, 480, 0, 0) != 0) {
        set_color_depth(15);
        if (set_gfx_mode(GFX_DIRECTX_WIN, 640, 480, 0, 0) != 0) {
           /* 640x480 direct drawing mode not supported */
           goto Error;
        }
     }
     /* ok, we are in direct drawing mode */
  }
  ```

  Note that, mainly for performance reasons, this driver requires the width of the screen to be a multiple of 4.
- GFX_DIRECTX_OVL
  The DirectX overlay driver. It uses special hardware features to run your program in a windowed mode: it doesn't work on all hardware, but performance is excellent on cards that are capable of it. It requires the color depth to be the same as that of the Windows desktop. In light of the limitation of desktop_color_depth() (see above), the reliable way of setting the overlay driver on desktops reported as 16-bit is:

  ```
  if (desktop_color_depth() == 16) {
     set_color_depth(16);
     if (set_gfx_mode(GFX_DIRECTX_OVL, 640, 480, 0, 0) != 0) {
        set_color_depth(15);
        if (set_gfx_mode(GFX_DIRECTX_OVL, 640, 480, 0, 0) != 0) {
           /* 640x480 overlay driver not supported */
           goto Error;
  ```

```
            }
         }
         /* ok, the 640x480 overlay driver is running */
      }
```

- GFX_GDI
  The windowed GDI driver. It is extremely slow, but is guaranteed to work on all hardware, so it can be useful for situations where you want to run in a window and don't care about performance. Note that this driver features a hardware mouse cursor emulation in order to speed up basic mouse operations (like GUI operations).

See also:

See Section 8.6 [set_gfx_mode], page 70.

35.2 DIGI_*/Windows

`Drivers DIGI_*/Windows`

The Windows sound functions support the following digital soundcards:

```
DIGI_AUTODETECT      - let Allegro pick a digital sound driver
DIGI_NONE            - no digital sound
DIGI_DIRECTX(n)      - use DirectSound device #n (zero-based) with
                       direct mixing
DIGI_DIRECTAMX(n)    - use DirectSound device #n (zero-based) with
                       Allegro mixing
DIGI_WAVOUTID(n)     - high (n=0) or low (n=1) quality WaveOut device
```

See also:

See Section 23.1 [detect_digi_driver], page 157.

See Section 23.5 [install_sound], page 160.

See Section 27.1 [install_sound_input], page 177.

35.3 MIDI_*/Windows

`Drivers MIDI_*/Windows`

The Windows sound functions support the following MIDI soundcards:

```
MIDI_AUTODETECT      - let Allegro pick a MIDI sound driver
MIDI_NONE            - no MIDI sound
MIDI_WIN32MAPPER     - use win32 MIDI mapper
MIDI_WIN32(n)        - use win32 device #n (zero-based)
MIDI_DIGMID          - sample-based software wavetable player
```

The following functions provide a platform specific interface to seamlessly integrate Allegro into general purpose Win32 programs. To use these routines, you must include winalleg.h after other Allegro headers.

See also:

See Section 23.2 [detect_midi_driver], page 158.

See Section 23.5 [install_sound], page 160.

See Section 27.1 [install_sound_input], page 177.

35.4 win_get_window

`HWND win_get_window(void);`

Retrieves a handle to the window used by Allegro. Note that Allegro uses an underlying window even though you don't set any graphics mode, unless you have installed the neutral system driver (SYSTEM_NONE).

35.5 win_set_window

`void win_set_window(HWND wnd);`

Registers an user-created window to be used by Allegro. This function is meant to be called before initialising the library with allegro_init() or installing the autodetected system driver (SYSTEM_AUTODETECT). It lets you attach Allegro to any already existing window and prevents the library from creating its own, thus leaving you total control over the window; in particular, you are responsible for processing the events as usual (Allegro will automatically monitor a few of them, but will not filter out any of them). You can then use every component of the library (mouse, keyboard, sound, timers and so on) except the graphics subsystem, bearing in mind that some Allegro functions are blocking (e.g readkey() if the key buffer is empty) and thus must be carefully manipulated by the window thread.

However you can also call it after the library has been initialised, provided that no graphics mode is set. In this case the keyboard, mouse, sound and sound recording modules will be restarted.

Passing NULL instructs Allegro to switch back to its built-in window if an user-created window was registered, or to request a new handle from Windows for its built-in window if this was already in use.

35.6 win_set_wnd_create_proc

`void win_set_wnd_create_proc(HWND (*proc)(WNDPROC));`

Registers an user-defined procedure to be used by Allegro for creating its window. This function must be called *before* initializing the library with allegro_init() or installing the autodetected system driver (SYSTEM_AUTODETECT). It lets you customize Allegro's window but only by its creation: unlike with win_set_window(), you have no control over the window once it has been created (in particular, you are not responsible for processing the events). The registered function will be passed a window procedure (WNDPROC object) that it must make the procedure of the new window of and it must return a handle to the new window. You can then use the full-featured library in the regular way.

35.7 win_get_dc

```
HDC win_get_dc(BITMAP *bmp);
```

Retrieves a handle to the device context of a DirectX video or system bitmap.

35.8 win_release_dc

```
void win_release_dc(BITMAP *bmp, HDC dc);
```

Releases a handle to the device context of the bitmap that was previously retrieved with win_get_dc().

35.9 GDI routines

The following GDI routines are a very platform specific thing, to allow drawing Allegro memory bitmaps onto a Windows device context. When you want to use this, you'll have to install the neutral system driver (SYSTEM_NONE) or attach Allegro to an external window with win_set_window().

There are two ways to draw your Allegro bitmaps to the Windows GDI. When you are using static bitmaps (for example just some pictures loaded from a datafile), you can convert them to DDB (device-dependent bitmaps) with convert_bitmap_to_hbitmap() and then just use Win32's BitBlt() to draw it.

When you are using dynamic bitmaps (for example some things which react to user input), it's better to use set_palette_to_hdc() and blit_to_hdc() functions, which work with DIB (device-independent bitmaps).

There are also functions to blit from a device context into an Allegro BITMAP, so you can do things like screen capture.

All the drawing and conversion functions use the current palette as a color conversion table. You can alter the current palette with the set_palette_to_hdc() or select_palette() functions. Warning: when the GDI system color palette is explicitly changed, (by another application, for example) the current Allegro palette is not updated along with it!

To use these routines, you must include winalleg.h after Allegro headers.

35.10 set_gdi_color_format

```
void set_gdi_color_format(void);
```

Tells Allegro to use the GDI color layout for truecolor images. This is optional, but it will make the conversions work faster. If you are going to call this, you should do it right after initialising Allegro and before creating any graphics.

35.11 set_palette_to_hdc

```
void set_palette_to_hdc(HDC dc, PALETTE pal);
```

Selects and realizes an Allegro palette on the specified device context.

35.12 convert_palette_to_hpalette

```
HPALETTE convert_palette_to_hpalette(PALETTE pal);
```

Converts an Allegro palette to a Windows palette and returns a handle to it. You should call DeleteObject() when you no longer need it.

See also:
See Section 35.13 [convert_hpalette_to_palette], page 240.

35.13 convert_hpalette_to_palette

```
void convert_hpalette_to_palette(HPALETTE hpal, PALETTE pal);
```

Converts a Windows palette to an Allegro palette.

See also:
See Section 35.12 [convert_palette_to_hpalette], page 239.

35.14 convert_bitmap_to_hbitmap

```
HBITMAP convert_bitmap_to_hbitmap(BITMAP *bitmap);
```

Converts an Allegro memory bitmap to a Windows DDB and returns a handle to it. This bitmap uses its own memory, so you can destroy the original bitmap without affecting the converted one. You should call DeleteObject() when you no longer need this bitmap.

See also:
See Section 35.15 [convert_hbitmap_to_bitmap], page 240.

35.15 convert_hbitmap_to_bitmap

```
BITMAP *convert_hbitmap_to_bitmap(HBITMAP bitmap);
```

Creates an Allegro memory bitmap from a Windows DDB.

See also:
See Section 35.14 [convert_bitmap_to_hbitmap], page 240.

35.16 draw_to_hdc

```
void draw_to_hdc(HDC dc, BITMAP *bitmap, int x, int y);
```

Draws an entire Allegro bitmap to a Windows device context, using the same parameters as the draw_sprite() function.

See also:
See Section 35.17 [blit_to_hdc], page 241.
See Section 35.18 [stretch_blit_to_hdc], page 241.
See Section 14.8 [draw_sprite], page 114.

35.17 blit_to_hdc

`void blit_to_hdc(BITMAP *bitmap, HDC dc, int sx, sy, dx, dy, w, h);`

Blits an Allegro memory bitmap to a Windows device context, using the same parameters as the blit() function.

See also:

See Section 35.16 [draw_to_hdc], page 240.

See Section 35.18 [stretch_blit_to_hdc], page 241.

See Section 35.19 [blit_from_hdc], page 241.

See Section 14.4 [blit], page 112.

35.18 stretch_blit_to_hdc

`void stretch_blit_to_hdc(BITMAP *bitmap, HDC dc, int sx, sy, sw, sh, int dx, dy, dw, dh);`

Blits an Allegro memory bitmap to a Windows device context, using the same parameters as the stretch_blit() function.

See also:

See Section 35.16 [draw_to_hdc], page 240.

See Section 35.17 [blit_to_hdc], page 241.

See Section 35.20 [stretch_blit_from_hdc], page 241.

See Section 14.5 [stretch_blit], page 113.

35.19 blit_from_hdc

`void blit_from_hdc(HDC hdc, BITMAP *bitmap, int sx, sy, dx, dy, w, h);`

Blits from a Windows device context to an Allegro memory bitmap, using the same parameters as the blit() function. See stretch_blit_from_hdc() for details.

See also:

See Section 35.20 [stretch_blit_from_hdc], page 241.

See Section 35.17 [blit_to_hdc], page 241.

See Section 14.4 [blit], page 112.

35.20 stretch_blit_from_hdc

`void stretch_blit_from_hdc(HDC hcd, BITMAP *bitmap, int sx, sy, sw, sh, int dx, dy, dw, dh);`

Blits from a Windows device context to an Allegro memory bitmap, using the same parameters as the stretch_blit() function. It uses the current Allegro palette and does conversion to this palette, regardless of the current DC palette.

So if you are blitting from 8 bit mode, you should first set the DC palette with the set_palette_to_hdc() function.

See also:
See Section 35.19 [blit_from_hdc], page 241.
See Section 35.18 [stretch_blit_to_hdc], page 241.
See Section 14.5 [stretch_blit], page 113.

36 Unix specifics

In order to locate things like the config and translation files, Allegro needs to know the path to your executable. Since there is no standard way to find that, it needs to capture a copy of your argv[] parameter, and it does this with some preprocessor macro trickery. Unfortunately it can't quite pull this off without a little bit of your help, so you will have to write END_OF_MAIN() right after your main() function. Pretty easy, really, and if you forget, you'll get a nice linker error about a missing _mangled_main function to remind you :-)

36.1 GFX_*/Linux

`Drivers GFX_*/Linux`

When running in Linux console mode, Allegro supports the following card parameters for the set_gfx_mode() function:

- GFX_TEXT
  Return to text mode.
- GFX_AUTODETECT
  Let Allegro pick an appropriate graphics driver.
- GFX_AUTODETECT_FULLSCREEN
  Autodetects a graphics driver, but will only use fullscreen drivers, failing if these are not available on current platform.
- GFX_AUTODETECT_WINDOWED
  Same as above, but uses only windowed drivers. This will always fail under Linux console mode.
- GFX_SAFE
  Special driver for when you want to reliably set a graphics mode and don't really care what resolution or color depth you get. See the set_gfx_mode() documentation for details.
- GFX_FBCON
  Use the framebuffer device (eg. /dev/fb0). This requires you to have framebuffer support compiled into your kernel, and correctly configured for your hardware. It is currently the only console mode driver that will work without root permissions, unless you are using a development version of SVGAlib.
- GFX_VBEAF
  Use a VBE/AF driver (vbeaf.drv), assuming that you have installed one

which works under Linux (currently only two of the FreeBE/AF project drivers are capable of this: I don't know about the SciTech ones). VBE/AF requires root permissions, but is currently the only Linux driver which supports hardware accelerated graphics.

- GFX_SVGALIB
  Use the SVGAlib library for graphics output. This requires root permissions if your version of SVGAlib requires them.
- GFX_VGA
  GFX_MODEX
  Use direct hardware access to set standard VGA or mode-X resolutions, supporting the same modes as in the DOS versions of these drivers. Requires root permissions.

See also:

See Section 8.6 [set_gfx_mode], page 70.

See Section 36.2 [GFX_*/X], page 243.

36.2 GFX_*/X

`Drivers GFX_*/X`

When running in X mode, Allegro supports the following card parameters for the set_gfx_mode() function:

- GFX_TEXT
  This closes any graphic mode previously opened with set_gfx_mode.
- GFX_AUTODETECT
  Let Allegro pick an appropriate graphics driver.
- GFX_AUTODETECT_FULLSCREEN
  Autodetects a graphics driver, but will only use fullscreen drivers, failing if these are not available on current platform.
- GFX_AUTODETECT_WINDOWED
  Same as above, but uses only windowed drivers.
- GFX_SAFE
  Special driver for when you want to reliably set a graphics mode and don't really care what resolution or color depth you get. See the set_gfx_mode() documentation for details.
- GFX_XWINDOWS
  The standard X graphics driver. This should work on any Unix system, and can operate remotely. It does not require root permissions.
- GFX_XWINDOWS_FULLSCREEN
  The same as above, but while GFX_XWINDOWS runs windowed, this one uses the XF86VidMode extension to make it run in fullscreen mode even without root permissions. You're still using the standard X protocol though, so expect the same low performances as with the windowed driver version.

- GFX_XDGA
  Use the XFree86 DGA 1.0 extension to write directly to the screen surface. DGA is normally much faster than the standard X mode, but does not produce such well behaved windowed programs, and will not work remotely. This driver requires root permissions.
- GFX_XDGA_FULLSCREEN
  Like GFX_XDGA, but also changes the screen resolution so that it will run fullscreen. This driver requires root permissions.
- GFX_XDGA2
  Use new DGA 2.0 extension provided by XFree86 4.0.x. This will work in fullscreen mode, and it will support hardware acceleration if available. This driver requires root permissions.
- GFX_XDGA2_SOFT
  The same as GFX_XDGA2, but turns off hardware acceleration support. This driver requires root permissions.

See also:

See Section 8.6 [set_gfx_mode], page 70.

See Section 36.1 [GFX_*/Linux], page 242.

36.3 DIGI_*/Unix

`Drivers DIGI_*/Unix`

The Unix sound functions support the following digital soundcards:

```
DIGI_AUTODETECT      - let Allegro pick a digital sound driver
DIGI_NONE            - no digital sound
DIGI_OSS             - Open Sound System
DIGI_ESD             - Enlightened Sound Daemon
DIGI_ARTS            - aRts (Analog Real-Time Synthesizer)
DIGI_ALSA            - ALSA sound driver
```

See also:

See Section 23.1 [detect_digi_driver], page 157.

See Section 23.5 [install_sound], page 160.

See Section 27.1 [install_sound_input], page 177.

36.4 MIDI_*/Unix

`Drivers MIDI_*/Unix`

The Unix sound functions support the following MIDI soundcards:

```
MIDI_AUTODETECT      - let Allegro pick a MIDI sound driver
MIDI_NONE            - no MIDI sound
```

```
          MIDI_OSS            - Open Sound System
          MIDI_DIGMID         - sample-based software wavetable player
          MIDI_ALSA           - ALSA RawMIDI driver
```

See also:
See Section 23.2 [detect_midi_driver], page 158.
See Section 23.5 [install_sound], page 160.
See Section 27.1 [install_sound_input], page 177.

37 BeOS specifics

37.1 GFX_*/BeOS

`Drivers GFX_*/BeOS`

BeOS Allegro supports the following card parameters for the set_gfx_mode() function:

- GFX_TEXT
  This closes any graphic mode previously opened with set_gfx_mode.
- GFX_AUTODETECT
  Let Allegro pick an appropriate graphics driver.
- GFX_AUTODETECT_FULLSCREEN
  Autodetects a graphics driver, but will only use fullscreen drivers, failing if these are not available on current platform.
- GFX_AUTODETECT_WINDOWED
  Same as above, but uses only windowed drivers.
- GFX_SAFE
  Special driver for when you want to reliably set a graphics mode and don't really care what resolution. See the set_gfx_mode() documentation for details.
- GFX_BEOS_FULLSCREEN
  Fullscreen exclusive mode. Supports only resolutions higher or equal to 640x480, and uses hardware acceleration if available.
- GFX_BEOS_FULLSCREEN_SAFE
  Works the same as GFX_FULLSCREEN, but disables acceleration.
- GFX_BEOS_WINDOWED
  Fast windowed mode using the BDirectWindow class. Not all graphics cards support this.

See also:
See Section 8.6 [set_gfx_mode], page 70.

37.2 DIGI_*/BeOS

`Drivers DIGI_*/BeOS`

The BeOS sound functions support the following digital soundcards:

```
DIGI_AUTODETECT      - let Allegro pick a digital sound driver
DIGI_NONE            - no digital sound
DIGI_BEOS            - BeOS digital output
```

See also:

See Section 23.1 [detect_digi_driver], page 157.

See Section 23.5 [install_sound], page 160.

See Section 27.1 [install_sound_input], page 177.

37.3 MIDI_*/BeOS

`Drivers MIDI_*/BeOS`

The BeOS sound functions support the following MIDI soundcards:

```
MIDI_AUTODETECT      - let Allegro pick a MIDI sound driver
MIDI_NONE            - no MIDI sound
MIDI_BEOS            - BeOS MIDI output
MIDI_DIGMID          - sample-based software wavetable player
```

See also:
See Section 23.2 [detect_midi_driver], page 158.
See Section 23.5 [install_sound], page 160.
See Section 27.1 [install_sound_input], page 177.

38 QNX specifics

38.1 GFX_*/QNX

`Drivers GFX_*/QNX`

QNX Allegro supports the following card parameters for the set_gfx_mode() function:

- GFX_TEXT
  This closes any graphic mode previously opened with set_gfx_mode.
- GFX_AUTODETECT
  Let Allegro pick an appropriate graphics driver.
- GFX_AUTODETECT_FULLSCREEN
  Autodetects a graphics driver, but will only use fullscreen drivers, failing if these are not available on current platform.

- GFX_AUTODETECT_WINDOWED
  Same as above, but uses only windowed drivers.
- GFX_SAFE
  Special driver for when you want to reliably set a graphics mode and don't really care what resolution. See the set_gfx_mode() documentation for details.
- GFX_PHOTON_DIRECT
  Fullscreen exclusive mode through Photon.
- GFX_PHOTON
  Windowed mode in a Photon window. Note that, mainly for performance reasons, this driver requires the width of the screen to be a multiple of 4.

See also:

See Section 8.6 [set_gfx_mode], page 70.

38.2 DIGI_*/QNX

`Drivers DIGI_*/QNX`

The QNX sound functions support the following digital soundcards:

```
DIGI_AUTODETECT      - let Allegro pick a digital sound driver
DIGI_NONE            - no digital sound
DIGI_ALSA            - ALSA sound driver
```

See also:

See Section 23.1 [detect_digi_driver], page 157.

See Section 23.5 [install_sound], page 160.

See Section 27.1 [install_sound_input], page 177.

38.3 MIDI_*/QNX

`Drivers MIDI_*/QNX`

The QNX sound functions support the following MIDI soundcards:

```
MIDI_AUTODETECT      - let Allegro pick a MIDI sound driver
MIDI_NONE            - no MIDI sound
MIDI_ALSA            - ALSA RawMIDI driver
MIDI_DIGMID          - sample-based software wavetable player
```

The following functions provide a platform specific interface to seamlessly integrate Allegro into general purpose QNX programs. To use these routines, you must include qnxalleg.h after other Allegro headers.

See also:

See Section 23.2 [detect_midi_driver], page 158.

See Section 23.5 [install_sound], page 160.

See Section 27.1 [install_sound_input], page 177.

38.4 qnx_get_window

```
PtWidget_t qnx_get_window(void);
```

Retrieves a handle to the window used by Allegro. Note that Allegro uses an underlying window even though you don't set any graphics mode, unless you have installed the neutral system driver (SYSTEM_NONE).

39 Differences between platforms

Here's a quick summary of things that may cause problems when moving your code from one platform to another (you can find a more detailed version of this in the docs section of the Allegro website).

The Windows and Unix versions require you to write END_OF_MAIN() after your main() function, which is used to magically turn an ANSI C style main() into a Windows style WinMain(), and so that the Unix code can grab a copy of your argv[] parameter.

On many platforms Allegro runs very slowly if you rely on it in order to automatically lock bitmaps when drawing onto them. For good performance, you need to call acquire_bitmap() and release_bitmap() yourself, and try to keep the amount of locking to a minimum.

The Windows version may lose the contents of video memory if the user switches away from your program, so you need to deal with that.

None of the currently supported platforms require input polling, but it is possible that some future ones might, so if you want to ensure 100% portability of your program, you should call poll_mouse() and poll_keyboard() in all the relevant places.

Allegro defines a number of standard macros that can be used to check various attributes of the current platform:

ALLEGRO_PLATFORM_STR
Text string containing the name of the current platform.

ALLEGRO_DOS
ALLEGRO_DJGPP
ALLEGRO_WATCOM
ALLEGRO_WINDOWS
ALLEGRO_MSVC
ALLEGRO_MINGW32
ALLEGRO_BCC32
ALLEGRO_UNIX
ALLEGRO_LINUX
ALLEGRO_BEOS
ALLEGRO_QNX
ALLEGRO_GCC
Defined if you are building for a relevant system. Often several of these will apply, eg. DOS+Watcom, or Windows+GCC+MinGW32.

ALLEGRO_I386
ALLEGRO_BIG_ENDIAN
ALLEGRO_LITTLE_ENDIAN
Defined if you are building for a processor of the relevant type.

ALLEGRO_VRAM_SINGLE_SURFACE
Defined if the screen is a single large surface that is then partitioned into multiple video sub-bitmaps (eg. DOS), rather than each video bitmap being a totally unique entity (eg. Windows).

ALLEGRO_CONSOLE_OK
Defined if when you are not in a graphics mode, there is a text mode console that you can printf() to, and from which the user could potentially redirect stdout to capture it even while you are in a graphics mode. If this define is absent, you are running in an environment like Windows that has no stdout at all.

ALLEGRO_MAGIC_MAIN
Defined if Allegro uses a magic main, i.e takes over the main() entry point and turns it into a secondary entry point suited to its needs.

ALLEGRO_LFN
Non-zero if long filenames are supported, or zero if you are limited to 8.3 format (in the djgpp version, this is a variable depending on the runtime environment).

LONG_LONG
Defined to whatever represents a 64-bit "long long" integer for the current compiler, or not defined if that isn't supported.

OTHER_PATH_SEPARATOR
Defined to a path separator character other than a forward slash for platforms that use one (eg. a backslash under DOS and Windows), or defined to a forward slash if there is no other separator character.

DEVICE_SEPARATOR
Defined to the filename device separator character (a colon for DOS and Windows), or to zero if there are no explicit devices in paths (Unix).

Allegro also defines a number of standard macros that can be used to insulate you from some of the differences between systems:

USE_CONSOLE
If you define this prior to including Allegro headers, Allegro will be set up for building a console application rather than the default GUI program on some platforms (especially Windows).

INLINE
Use this in place of the regular "inline" function modifier keyword, and your code will work correctly on any of the supported compilers.

ZERO_SIZE_ARRAY(type, name)
Use this to declare zero-sized arrays in terminal position inside structures, like in the BITMAP structure. These arrays are effectively equivalent to the flexible array members of ISO C99.

40 Reducing your executable size

Some people complain that Allegro produces very large executables. This is certainly true: with the djgpp version, a simple "hello world" program will be about 200k, although the per-executable overhead is much less for platforms that support dynamic linking. But don't worry, Allegro takes up a relatively fixed amount of space, and won't increase as your program gets larger. As George Foot so succinctly put it, anyone who is concerned about the ratio between library and program code should just get to work and write more program code to catch up :-)

Having said that, there are several things you can do to make your programs smaller:

- For all platforms, you can use an executable compressor called UPX, which is available at http://upx.tsx.org/ . This usually manages a compression ratio of about 40%.
- When using djgpp: for starters, read the djgpp FAQ section 8.14, and take note of the -s switch. And don't forget to compile your program with optimisation enabled!
- If a DOS program is only going to run in a limited number of graphics modes, you can specify which graphics drivers you would like to include with the code:

  ```
  BEGIN_GFX_DRIVER_LIST
     driver1
     driver2
     etc...
  END_GFX_DRIVER_LIST
  ```

  where the driver names are any of the defines:

  ```
  GFX_DRIVER_VBEAF
  GFX_DRIVER_VGA
  GFX_DRIVER_MODEX
  GFX_DRIVER_VESA3
  GFX_DRIVER_VESA2L
  GFX_DRIVER_VESA2B
  GFX_DRIVER_XTENDED
  GFX_DRIVER_VESA1
  ```

  This construct must be included in only one of your C source files. The ordering of the names is important, because the autodetection routine works down from the top of the list until it finds the first driver that is able to support the requested mode. I suggest you stick to the default ordering given above, and simply delete whatever entries you aren't going to use.
- If your DOS program doesn't need to use all the possible color depths, you can specify which pixel formats you want to support with the code:

  ```
  BEGIN_COLOR_DEPTH_LIST
     depth1
     depth2
     etc...
  END_COLOR_DEPTH_LIST
  ```

where the color depth names are any of the defines:

```
COLOR_DEPTH_8
COLOR_DEPTH_15
COLOR_DEPTH_16
COLOR_DEPTH_24
COLOR_DEPTH_32
```

Removing any of the color depths will save quite a bit of space, with the exception of the 15 and 16 bit modes: these share a great deal of code, so if you are including one of them, there is no reason not to use both. Be warned that if you try to use a color depth which isn't in this list, your program will crash horribly!

- In the same way as the above, you can specify which DOS sound drivers you want to support with the code:

```
BEGIN_DIGI_DRIVER_LIST
   driver1
   driver2
   etc...
END_DIGI_DRIVER_LIST
```

  using the digital sound driver defines:

```
DIGI_DRIVER_SOUNDSCAPE
DIGI_DRIVER_AUDIODRIVE
DIGI_DRIVER_WINSOUNDSYS
DIGI_DRIVER_SB
```

  and for the MIDI music:

```
BEGIN_MIDI_DRIVER_LIST
   driver1
   driver2
   etc...
END_MIDI_DRIVER_LIST
```

  using the MIDI driver defines:

```
MIDI_DRIVER_AWE32
MIDI_DRIVER_DIGMID
MIDI_DRIVER_ADLIB
MIDI_DRIVER_MPU
MIDI_DRIVER_SB_OUT
```

  If you are going to use either of these sound driver constructs, you must include both.

- Likewise for the DOS joystick drivers, you can declare an inclusion list:

```
BEGIN_JOYSTICK_DRIVER_LIST
   driver1
```

```
      driver2
      etc...
   END_JOYSTICK_DRIVER_LIST
```

using the joystick driver defines:

```
   JOYSTICK_DRIVER_WINGWARRIOR
   JOYSTICK_DRIVER_SIDEWINDER
   JOYSTICK_DRIVER_GAMEPAD_PRO
   JOYSTICK_DRIVER_GRIP
   JOYSTICK_DRIVER_STANDARD
   JOYSTICK_DRIVER_SNESPAD
   JOYSTICK_DRIVER_PSXPAD
   JOYSTICK_DRIVER_N64PAD
   JOYSTICK_DRIVER_DB9
   JOYSTICK_DRIVER_TURBOGRAFX
   JOYSTICK_DRIVER_IFSEGA_ISA
   JOYSTICK_DRIVER_IFSEGA_PCI
   JOYSTICK_DRIVER_IFSEGA_PCI_FAST
```

The standard driver includes support for the dual joysticks, increased numbers of buttons, Flightstick Pro, and Wingman Extreme, because these are all quite minor variations on the basic code.

- If you are _really_ serious about this size, thing, have a look at the top of include/allegro/alconfig.h and you will see the lines:

```
   #define ALLEGRO_COLOR8
   #define ALLEGRO_COLOR16
   #define ALLEGRO_COLOR24
   #define ALLEGRO_COLOR32
```

  If you comment out any of these definitions and then rebuild the library, you will get a version without any support for the absent color depths, which will be even smaller than using the DECLARE_COLOR_DEPTH_LIST() macro. Removing the ALLEGRO_COLOR16 define will get rid of the support for both 15 and 16 bit hicolor modes, since these share a lot of the same code.

Note: the aforementioned methods for removing unused hardware drivers only apply to statically linked versions of the library, eg. DOS. On Windows and Unix platforms, you can build Allegro as a DLL or shared library, which prevents these methods from working, but saves so much space that you probably won't care about that. Removing unused color depths from alconfig.h will work on any platform, though.

If you are distributing a copy of the setup program along with your game, you may be able to get a dramatic size reduction by merging the setup code into your main program, so that only one copy of the Allegro routines will need to be linked. See setup.txt for details. In the djgpp version, after compressing the executable, this will probably save you about 200k compared to having two separate programs for the setup and the game itself.

41 Debugging

There are three versions of the Allegro library: the normal optimised code, one with extra debugging support, and a profiling version. See the platform specific readme files for information about how to install and link with these alternative libs. Although you will obviously want to use the optimised library for the final version of your program, it can be very useful to link with the debug lib while you are working on it, because this will make debugging much easier, and includes assert tests that will help to locate errors in your code at an earlier stage. Allegro also contains some debugging helper functions:

41.1 al_assert

`void al_assert(const char *file, int line);`

Raises an assert for an error at the specified file and line number. The file parameter is always given in ASCII format. If you have installed a custom assert handler it uses that, or if the environment variable ALLEGRO_ASSERT is set it writes a message into the file specified by the environment, otherwise it aborts the program with an error message. You will usually want to use the ASSERT() macro instead of calling this function directly.

See also:

See Section 41.3 [ASSERT], page 253.

See Section 41.2 [al_trace], page 253.

See Section 41.5 [register_assert_handler], page 254.

41.2 al_trace

`void al_trace(const char *msg, ...);`

Outputs a debugging trace message, using a printf() format string given in ASCII. If you have installed a custom trace handler it uses that, or if the environment variable ALLEGRO_TRACE is set it writes into the file specified by the environment, otherwise it writes the message to "allegro.log" in the current directory. You will usually want to use the TRACE() macro instead of calling this function directly.

See also:

See Section 41.4 [TRACE], page 254.

See Section 41.1 [al_assert], page 253.

See Section 41.6 [register_trace_handler], page 254.

41.3 ASSERT

`void ASSERT(condition);`

Debugging helper macro. Normally compiles away to nothing, but if you defined the preprocessor symbol DEBUGMODE before including Allegro headers, it will check the supplied condition and call al_assert() if it fails.

See also:
See Section 41.1 [al_assert], page 253.
See Section 41.4 [TRACE], page 254.
See Section 41.5 [register_assert_handler], page 254.

41.4 TRACE

```
void TRACE(char *msg, ...);
```

Debugging helper macro. Normally compiles away to nothing, but if you defined the preprocessor symbol DEBUGMODE before including Allegro headers, it passes the supplied message given in ASCII format to al_trace().

See also:
See Section 41.2 [al_trace], page 253.
See Section 41.3 [ASSERT], page 253.
See Section 41.6 [register_trace_handler], page 254.

41.5 register_assert_handler

```
void register_assert_handler(int (*handler)(const char *msg));
```

Supplies a custom handler function for dealing with assert failures. Your callback will be passed a formatted error message in ASCII, and should return non-zero if it has processed the error, or zero to continue with the default actions. You could use this to ignore assert failures, or to display the error messages on a graphics mode screen without aborting the program.

See also:
See Section 41.1 [al_assert], page 253.
See Section 41.3 [ASSERT], page 253.
See Section 41.6 [register_trace_handler], page 254.

41.6 register_trace_handler

```
void register_trace_handler(int (*handler)(const char *msg));
```

Supplies a custom handler function for dealing with trace output. Your callback will be passed a formatted error message in ASCII, and should return non-zero if it has processed the message, or zero to continue with the default actions. You could use this to ignore trace output, or to display the messages on a second monochrome monitor, etc.

See also:
See Section 41.2 [al_trace], page 253.
See Section 41.4 [TRACE], page 254.

See Section 41.5 [register_assert_handler], page 254.

42 Makefile targets

There are a number of options that you can use to control exactly how Allegro will be compiled. On Unix platforms you do this by passing arguments to the configure script (run "configure –help" for a list), but on other platforms you can set the environment variables:

- DEBUGMODE=1
  Selects a debug build, rather than the normal optimised version.
- DEBUGMODE=2
  Selects a build intended to debug Allegro itself, rather than the normal optimised version.
- PROFILEMODE=1
  Selects a profiling build, rather than the normal optimised version.
- WARNMODE=1
  Selects strict compiler warnings. If you are planning to work on Allegro yourself, rather than just using it in your programs, you should be sure to have this mode enabled.
- STATICLINK=1 (MSVC and Mingw32 only)
  Link as a static library, rather than the default DLL.
- TARGET_ARCH_COMPAT=[cpu] (implemented for most GNU platforms)
  This option will optimize for the given processor while maintaining compatibility with older processors. Example: set TARGET_ARCH_COMPAT=i586
- TARGET_ARCH_EXCL=[cpu] (implemented for most GNU platforms)
  This option will optimize for the given processor. Please note that using it will cause the code to *NOT* run on older processors. Example: set TARGET_ARCH_EXCL=i586
- TARGET_OPTS=[opts] (implemented for most GNU platforms)
  This option allows you to customize general compiler optimisations.
- CROSSCOMPILE=1 (djgpp only)
  Allows you to build the djgpp library under Linux, using djgpp as a cross-compiler.
- ALLEGRO_USE_C=1 (djgpp only)
  Allows you to build the djgpp library using C drawing code instead of the usual asm routines. This is only really useful for testing, since the asm version is faster.

If you only want to recompile a specific test program or utility, you can specify it as an argument to make, eg. "make demo" or "make grabber". The makefiles also provide some special pseudo-targets:

- 'default'
  The normal build process. Compiles the current library version (one of optimised, debugging, or profiling, selected by the above environment variables), builds the test and example programs, and converts the documentation files.
- 'all'
  Compiles all three library versions (optimised, debugging, and profiling), builds the test and example programs, and converts the documentation files.

- 'lib'
  Compiles the current library version (one of optimised, debugging, or profiling, selected by the above environment variables).
- 'install'
  Copies the current library version (one of optimised, debugging, or profiling, selected by the above environment variables), into your compiler lib directory, recompiling it as required, and installs the Allegro headers.
- 'installall'
  Copies all three library versions (optimised, debugging, and profiling), into your compiler lib directory, recompiling them as required, and installs the Allegro headers.
- 'uninstall'
  Removes the Allegro library and headers from your compiler directories.
- 'docs'
  Converts the documentation files from the ._tx sources.
- 'docs-dvi' (Unix only)
  Creates the allegro.dvi device independent documentation file. This is not a default target, since you need the texi2dvi tool to create it. The generated file is especially prepared to be printed on paper.
- 'docs-ps' or 'docs-gzipped-ps' (Unix only)
  Creates a Postcript file from the previously generated DVI file. This is not a default target, since you need the texi2dvi and dvips tools to create it. The second target compresses the generated Postscript file. The generated file is especially prepared to be printed on paper.
- 'install-man' or 'install-gzipped-man' (Unix only)
  This generates Unix man pages for each Allegro function or variable and installs them. The second target compresses the manual pages after installing them.
- 'install-info' or 'install-gzipped-info' (Unix only)
  Converts the documentation to Info format and installs it. The second target compresses the info file after installing it.
- 'clean'
  Removes generated object and library files, either to recover disk space or to force a complete rebuild the next time you run make. This target is designed so that if you run a "make install" followed by "make clean", you will still have a functional version of Allegro.
- 'distclean'
  Like "make clean", but more so. This removes all the executable files and the documentation, leaving you with only the same files that are included when you unzip a new Allegro distribution.
- 'veryclean'
  Use with extreme caution! This target deletes absolutely all generated files, including some that may be non-trivial to recreate. After you run "make veryclean", a simple rebuild will not work: at the very least you will have to run "make depend", and perhaps also fixdll.bat if you are using the Windows library. These targets make use of non-standard tools like SED, so unless you know what you are doing and have all this stuff installed, you should not use them.

- ’depend’
  Regenerates the dependency files (obj/*/makefile.dep). You need to run this after `"make veryclean"`, or whenever you add new headers to the Allegro sources.
- ’compress’ (djgpp, Mingw32 and MSVC only)
  Uses the DJP or UPX executable compressors (whichever you have installed) to compress the example executables and utility programs, which can recover a significant amount of disk space.

43 Conclusion

All good things must come to an end. Writing documentation is not a good thing, though, and that means it goes on for ever. There is always something I’ve forgotten to explain, or some essential detail I’ve left out, but for now you will have to make do with this. Feel free to ask if you can’t figure something out.

Enjoy. I hope you find some of this stuff useful.

By Shawn Hargreaves.

http://alleg.sourceforge.net/

Table of Contents