Glk API Specification

API version 0.7.4

Andrew Plotkin <erkyrath@eblong.com>

The API *described* by this document is an idea, not an expression of an idea, and is therefore not copyrightable. Anyone is free to write programs that use the Glk API or libraries that implement it.

This document and further Glk information can be found at: <http://eblong.com/zarf/glk/>

0. Introduction

0.1. What Glk Is

Glk defines a portable API (programming interface) for applications with text UIs (user interfaces.) It was primarily designed for interactive fiction, but it should be suitable for many interactive text utilities, particularly those based on a command line.

Rather than go into a detailed explanation of what that means, let me give examples from the world of text adventures. TADS, Glulx, and Infocom’s Z-machine have nearly identical interface capabilities; each allows a program to...

- print an indefinite stream of text into an output buffer, with some style control
- input a line of text
- display a few lines of text in a small separate window
- store information in a file, or read it in

and so on. However, the implementation of these capabilities vary widely between platforms and operating systems. Furthermore, this variance is transparent to the program (the adventure game.) The game does not care whether output is displayed via a character terminal emulator or a GUI window; nor whether input uses Mac-style mouse editing or EMACS-style control key editing.

On the third hand, the user is likely to care deeply about these interface decisions. This is why there are Mac-native interpreters on Macintoshes, stylus and touch-screen interpreters on mobile devices, and so on — and (ultimately) why there are Macintoshes and iPads and terminal window apps in the first place.

On the *fourth* hand, TADS and Inform are not alone; there is historically a large number of text adventure systems. Most are obsolete or effectively dead; but it is inevitable that more will appear. Users want each living system ported to all the platforms in use. Users also prefer these ports to use the same interface, as much as possible.

This all adds up to a pain in the ass.

Glk tries to draw a line between the parts of the text adventure world which are identical on all IF systems, and different on different operating systems, from the parts which are unique to each IF system but identical in all OSs. The border between these two worlds is the Glk API.

My hope is that a new IF system, or existing ones which are less-supported (Hugo, AGT, etc) can be written using Glk for all input and output function. The IF system would then be in *truly* portable C. On the other side of the line, there would be a Glk library for each operating system and interface (Macintosh, X-windows, curses-terminal, etc.) Porting the IF system to every platform would be trivial; compile the system, and link in the library.

Glk can also serve as a nice interface for applications other than games — data manglers, quick hacks, or anything else which would normally lack niceties such as editable input, macros, scrolling, or whatever is native to your machine's interface idiom.

0.2. What About the Virtual Machine?

You can think of Glk as an IF virtual machine, without the virtual machine part. The "machine" is just portable C code.

An IF virtual machine has been designed specifically to go along with Glk. This VM, called Glulx, uses Glk as its interface; each Glk call corresponds to an input/output opcode of the VM.

For more discussion of this approach, see section 11.3, "Glk and the Virtual Machine". Glulx is documented at <http://eblong.com/zarf/glulx/>.

Of course, Glk can be used with other IF systems. The advantage of Glulx is that it provides the game author with direct and complete access to the Glk API. Other IF systems typically have an built-in abstract I/O API, which maps only partially onto Glk. For these systems, Glk tends to be a least-common-denominator interface: highly portable, but not necessarily featureful. (Even if Glk has a feature, it may not be available through the layers of abstraction.)

0.3. What Does Glk Not Do?

Glk does not handle the things which should be handled by the program (or the IF system, or the virtual machine) which is linked to Glk. This means that Glk does not address

- parsing
- game object storage
- computation
- text compression

0.4. Conventions of This Document

This document defines the Glk API. I have tried to specify exactly what everything does, what is legal, what is illegal, and why.

Sections in square brackets *[like this]* are notes. They do not define anything; they clarify or explain what has already been defined. If there seems to be a conflict, ignore the note and follow the definition.

[Notes with the label "WORK" are things which I have not yet fully resolved. Your comments requested and welcome.]

This document is written for the point of view of the game programmer — the person who wants to use the Glk library to print text, input text, and so on. By saying what the Glk library does, of course, this document also defines the task of the Glk programmer — the person who wants to port the Glk library to a new platform or operating system. If the Glk library guarantees something, the game programmer can rely on it, and the Glk programmer is required to support it. Contrariwise, if the library does not guarantee something, the Glk programmer may handle it however he likes, and the game programmer must not rely on it. If something is illegal, the game programmer must not do it, and the Glk programmer is not required to worry about it. *[It is preferable, but not required, that the Glk library detect illegal requests and display error messages. The Glk library may simply crash when the game program does something illegal. This is why the game programmer must not do it. Right?]*

Hereafter, "Glk" or "the library" refers to the Glk library, and "the program" is the game program (or whatever) which is using the Glk library to print text, input text, or whatever. "You" are the person writing the program. "The player" is the person who will use the program/Glk library combination to actually play a game. Or whatever.

The Glk API is declared in a C header file called "glk.h". Please refer to that file when reading this one.

0.5. Credits

Glk has been a work of many years and many people. If I tried to list everyone who has offered comments and suggestions, I would immediately go blank, forget everyone's name, and become a mute hermit-like creature living in a train tunnel under Oakland. But I must thank those people who have written Glk libraries and linking systems: Matt Kimball, Ross Raszewski, David Kinder, John Elliott, Joe Mason, Stark Springs, and, er, anyone I missed. Look! A train!

Evin Robertson wrote the original proposal for the Glk Unicode functions, which I imported nearly verbatim into this document. Thank you.

1. Overall Structure

1.1. Your Program's Main Function

The top level of the program — the main() function in C, for example — belongs to Glk. *[This means that Glk isn't really a library. In a sense, you are writing a library, which is linked into Glk. This is bizarre to think about, so forget it.]*

You define a function called glk_main(), which the library calls to begin running your program. glk_main() should run until your program is finished, and then return.

Glk does all its user-interface work in a function called glk_select(). This function waits for an event — typically the player's input — and returns an structure representing that event. This means that your program must have an event loop. In the very simplest case, you could write

```
void glk_main()
{
    event_t ev;
    while (1) {
        glk_select(&ev);
        switch (ev.type) {
            default:
                /* do nothing */
                break;
        }
    }
}
```

This is a legal Glk-compatible program. As you might expect, it doesn't do anything. The player will see an empty window, which he can only stare at, or destroy in a platform-defined standard manner. *[Command-period on the Macintosh; a kill-window menu option in an X window manager; control-C in a curses terminal window.]*

[However, this program does not spin wildly and burn CPU time. The glk_select() function waits for an event it can return. Since it only returns events which you have requested, it will wait forever, and grant CPU time to other processes if that's meaningful on the player's machine.] [Actually, there are some events which are always reported. More may be defined in future versions of the Glk API. This is why the default response to an event is to do nothing. If you don't recognize the event, ignore it.]

1.2. Exiting Your Program

If you want to shut down your program in the middle of your glk_main() function, you can call glk_exit().

```
void glk_exit(void);
```

This function does not return.

If you print some text to a window and then shut down your program, you can assume that the player will be able to read it. Most likely the Glk library will give a "Hit any key to exit" prompt. (There are other possibilities, however. A terminal-window version of Glk might simply exit and leave the last screen state visible in the terminal window.)

[You should only *shut down your program with glk_exit() or by returning from your glk_main() function. If you call the ANSI exit() function, or other platform-native functions, bad things may happen. Some versions of the Glk library may be designed for multiple sessions, for example, and you would be cutting off all the sessions instead of just yours. You would probably also prevent final text from being visible to the player.]*

1.3. The Interrupt Handler

Most platforms have some provision for interrupting a program — command-period on the Macintosh, control-C in Unix, possibly a window manager menu item, or other possibilities. This can happen at any time, including while execution is nested inside one of your own functions, or inside a Glk library function.

If you need to clean up critical resources, you can specify an interrupt handler function.

```
void glk_set_interrupt_handler(void (*func)(void));
```

The argument you pass to glk_set_interrupt_handler() should be a pointer to a function which takes no argument and returns no result. If Glk receives an interrupt, and you have set an interrupt handler, your handler will be called, before the process is shut down.

Initially there is no interrupt handler. You can reset to not having any by calling glk_set_interrupt_handler(NULL).

If you call glk_set_interrupt_handler() with a new handler function while an older one is set, the new one replaces the old one. Glk does not try to queue interrupt handlers.

You should not try to interact with the player in your interrupt handler. Do not call glk_select() or glk_select_poll(). Anything you print to a window may not be visible to the player.

1.4. The Tick Thing

Many platforms have some annoying thing that has to be done every so often, or the gnurrs come from the voodvork out and eat your computer.

Well, not really. But you should call glk_tick() every so often, just in case. It may be necessary to yield time to other applications in a cooperative-multitasking OS, or to check for player interrupts in an infinite loop.

```
    void glk_tick(void);
```

This call is fast; in fact, on average, it does nothing at all. So you can call it often. *[In a virtual machine interpreter, once per opcode is appropriate. A more parsimonious approach would be once per branch and function call opcode; this guarantees it will be called inside loops. In a program with lots of computation, pick a comparable rate.]*

glk_tick() does not try to update the screen, or check for player input, or any other interface task. For that, you should call glk_select() or glk_select_poll(). See section 4, "Events".

[Captious critics have pointed out that in the sample program model.c, I do not call glk_tick() at all. This is because model.c has no heavy loops. It does a bit of work for each command, and then cycles back to the top of the event loop. The glk_select() call, of course, blocks waiting for input, so it does all the yielding and interrupt-checking one could imagine.]

[Basically, you must ensure there's some fixed upper bound on the amount of computation that can occur before a glk_tick() (or glk_select()) occurs. In a VM interpreter, where the VM code might contain an infinite loop, this is critical. In a C program, you can often eyeball it.]

[But the next version of model.c will have a glk_tick() in the ornate printing loop of verb_yada(). Just to make the point.]

1.5. Basic Types

For simplicity, all the arguments used in Glk calls are of a very few types.

- 32-bit unsigned integer. Unsigned integers are used wherever possible, which is nearly everywhere. This type is called "glui32".
- 32-bit signed integer. This type is called "glsi32". Rarely used.
- References to library objects. These are pointers to opaque C structures; each library will use different structures, so you can not and should not try to manipulate their contents. See section 1.6, "Opaque Objects".
- Pointer to one of the above types.
- Pointer to a structure which consists entirely of the above types.
- Unsigned char. This is used only for Latin-1 text characters; see section 2, "Character Encoding".

- Pointer to char. Sometimes this means a null-terminated string; sometimes an unterminated buffer, with length as a separate gui32 argument. The documentation says which.
- Pointer to void. When nothing else will do.

1.6. Opaque Objects

Glk keeps track of a few classes of special objects. These are opaque to your program; you always refer to them using pointers to opaque C structures.

Currently, these classes are:

- Windows: Screen panels, used to input or output information.
- Streams: Data streams, to which you can input or output text. *[There are file streams and window streams, since you can output data to windows or files.]*
- File references: Pointers to files in permanent storage. *[In Unix a file reference is a pathname; on the Mac, an FSSpec. Actually there's a little more information included, such as file type and whether it is a text or binary file.]*
- Sound channels: Audio output channels. *[Not all Glk libraries support sound.]*

[Note that there may be more object classes in future versions of the Glk API.]

When you create one of these objects, it is always possible that the creation will fail (due to lack of memory, or some other OS error.) When this happens, the allocation function will return NULL instead of a valid pointer. You should always test for this possibility.

NULL is never the identifier of any object (window, stream, file reference, or sound channel). The value NULL is often used to indicate "no object" or "nothing", but it is not a valid reference. If a Glk function takes an object reference as an argument, it is illegal to pass in NULL unless the function definition says otherwise.

The glk.h file defines types "winid_t", "strid_t", "frefid_t", "schanid_t" to store references. These are pointers to struct glk_window_struct, glk_stream_struct, glk_fileref_struct, and glk_schannel_struct respectively. It is, of course, illegal to pass one kind of pointer to a function which expects another.

[This is how you deal with opaque objects from a C program. If you are using Glk through a virtual machine, matters will probably be different. Opaque objects may be represented as integers, or as VM objects of some sort.]

1.6.1. Rocks

Every one of these objects (window, stream, file reference, or sound channel) has a "rock" value. This is simply a 32-bit integer value which you provide, for your own purposes, when you create the object. *[The library — so to speak — stuffs this value under a rock for safe-keeping, and gives it back to you when you ask for it.]*

[If you don't know what to use the rocks for, provide 0 and forget about it.]

1.6.2. Iterating Through Opaque Objects

For each class of opaque objects, there is an iterate function, which you can use to obtain a list of all existing objects of that class. It takes the form

```
CLASSid_t glk_CLASS_iterate(CLASSid_t obj, glui32 *rockptr);
```

...where CLASS represents one of the opaque object classes. *[So, at the current time, these are the* *functions glk_window_iterate(), glk_stream_iterate(), and glk_fileref_iterate(). There may be more classes in future versions of the spec; they all behave the same.]*

Calling glk_CLASS_iterate(NULL, r) returns the first object; calling glk_CLASS_iterate(obj, r) returns the next object, until there aren't any more, at which time it returns NULL.

The rockptr argument is a pointer to a location; whenever glk_CLASS_iterate() returns an object, the object's rock is stored in the location (*rockptr). If you don't want the rocks to be returned, you may set rockptr to NULL.

You usually use this as follows:

```
obj = glk_CLASS_iterate(NULL, NULL);
while (obj) {
    /* ...do something with obj... */
    obj = glk_CLASS_iterate(obj, NULL);
}
```

If you create or destroy objects inside this loop, obviously, the results are unpredictable. However it is always legal to call glk_CLASS_iterate(obj, r) as long as obj is a valid object id, or NULL.

The order in which objects are returned is entirely arbitrary. The library may even rearrange the order every time you create or destroy an object of the given class. As long as you do not create or destroy any object, the rule is that glk_CLASS_iterate(obj, r) has a fixed result, and iterating through the results as above will list every object exactly once.

1.7. The Gestalt System

The "gestalt" mechanism (cheerfully stolen from the Mac OS) is a system by which the Glk API can be upgraded without making your life impossible. New capabilities (graphics, sound, or so on) can be added without changing the basic specification. The system also allows for "optional" capabilities — those which not all Glk library implementations will support — and allows you to check for their presence without trying to infer them from a version number.

The basic idea is that you can request information about the capabilities of the API, by calling the gestalt functions:

```
glui32 glk_gestalt(glui32 sel, glui32 val);
glui32 glk_gestalt_ext(glui32 sel, glui32 val, glui32 *arr, glui32 arrlen);
```

The selector (the "sel" argument) tells which capability you are requesting information about; the other three arguments are additional information, which may or may not be meaningful. The arr and arrlen arguments of glk_gestalt_ext() are always optional; you may always pass NULL and 0, if you do not want whatever information they represent. glk_gestalt() is simply a shortcut for this; glk_gestalt(x, y) is exactly the same as glk_gestalt_ext(x, y, NULL, 0).

The critical point is that if the Glk library has never heard of the selector sel, it will return 0. It is *always* safe to call glk_gestalt(x, y) (or glk_gestalt_ext(x, y, NULL, 0)). Even if you are using an old library, which was compiled before the given capability was imagined, you can test for the capability by calling glk_gestalt(); the library will correctly indicate that it does not support it, by returning 0.

(It is also safe to call glk_gestalt_ext(x, y, z, zlen) for an unknown selector x, where z is not NULL, as long as z points at an array of at least zlen elements. The selector will be careful not to write beyond that point in the array, if it writes to the array at all.)

(If a selector does not use the second argument, you should always pass 0; do not assume that the second argument is simply ignored. This is because the selector may be extended in the future. You will continue to get the current behavior if you pass 0 as the second argument, but other values may produce other behavior.)

1.8. The Version Number

For an example of the gestalt mechanism, consider the selector gestalt_Version. If you do

```
   glui32 res;
   res = glk_gestalt(gestalt_Version, 0);
```

res will be set to a 32-bit number which encodes the version of the Glk spec which the library implements. The upper 16 bits stores the major version number; the next 8 bits stores the minor version number; the low 8 bits stores an even more minor version number, if any. *[So the version number 78.2.11 would be encoded as 0x004E020B.]*

The current Glk specification version is 0.7.3, so this selector will return 0x00000703.

```
   glui32 res;
   res = glk_gestalt_ext(gestalt_Version, 0, NULL, 0);
```

does exactly the same thing. Note that, in either case, the second argument is not used; so you should always pass 0 to avoid future surprises.

1.9. Other API Conventions

The glk.h header file is the same on all platforms, with the sole exception of the typedef of glui32 and glsi32. These will always be defined as 32-bit unsigned and signed integer types, which may be "long" or "int" or some other C definition.

Note that all constants are #defines. All functions are currently actual function declarations (as opposed to macros), but this may change in future Glk revisions. As in the standard C library, if Glk function is defined by a macro, an actual function of the same name will also be available.

Functions that return or generate boolean values will produce only 0 (FALSE) or 1 (TRUE). Functions that accept boolean arguments will accept any value, with zero indicating FALSE and nonzero indicating TRUE.

NULL (when used in this document) refers to the C null pointer. As stated above, it is illegal to pass NULL to a function which is expecting a valid object reference, unless the function definition says otherwise.

Some functions have pointer arguments, acting as "variable" or "reference" arguments; the function’s intent is to return some value in the space pointed to by the argument. Unless the function says otherwise, it is legal to pass NULL to indicate that you do not care about that value.

2. Character Encoding

Glk has two separate, but parallel, APIs for managing text input and output. The basic functions deals entirely in 8-bit characters; their arguments are arrays of bytes (octets). These functions all assume the Latin-1 character encoding. Equivalently, they may be said to use code points U+00..U+FF of <Unicode>.

Latin-1 is an 8-bit character encoding; it maps numeric codes in the range 0 to 255 into printed characters. The values from 32 to 126 are the standard printable ASCII characters (’ ’ to ’~’). Values 0 to 31 and 127 to 159 are reserved for control characters, and have no printed equivalent.

[Note that the basic Glk text API does not *use UTF-8, or any other Unicode character form. Each character is represented by a single byte — even characters in the 128..255 range.]*

The extended, or "Unicode", Glk functions deal entirely in 32-bit words. They take arrays of words, not bytes, as arguments. They can therefore cope with any Unicode code point. The extended functions have names ending in "_uni".

[Since these functions deal in arrays of 32-bit words, they can be said to use the UTF-32 character encoding form. (But not *the UTF-32 character encoding* scheme *— that’s a stream of bytes which must be interpreted in big-endian or little-endian mode. Glk Unicode functions operate on long integers, not bytes.) UTF-32 is also known as UCS-4, according to the Unicode spec (appendix C.2), modulo some semantic requirements which we will not deal with here. For practical purposes, we can ignore the whole encoding issue, and assume that we are dealing with sequences of numeric code points.]*

[Why not UTF-8? It is a reasonable bare-bones compression algorithm for Unicode character streams; but IF systems typically have their own compression models for text. Compositing the two ideas causes more problems than it solves. The other advantage of UTF-8 is that 7-bit ASCII is automatically valid UTF-8; but this is not compelling for IF systems, in which the compiler can be tasked with generating consistent textual data. And UTF-8 is a variable-width encoding. Nobody ever wept at the prospect of avoiding that kettle of eels.]

[What about bi-directional text? It’s a good idea, and may show up in future versions of this document. It is not in this version because we want to get something simple implemented soon. For the moment, print out all text in reading order (not necessarily left-to-right) and hope for the best. Current suggestions include a stylehint_Direction, which the game can set to indicate that text in the given style should be laid out right-to-left. Top-to-bottom (or bottom-to-top) may be desirable too. The direction stylehints might only apply to full paragraphs (like justification stylehints); or they might apply to any text, thus requiring the library to lay out "zig-zag" blocks. The possibilities remain to be explored. Page layout is hard.]

[Another possibility is to let the library determine the directionality of text from the character set. This is not impossible — MacOSX text widgets do it. It may be too difficult.]

[In the meantime, it is worth noting that the Windows Glk library does not *autodetect directionality, but the CheapGlk library running on MacOSX does. Therefore, there is no platform-independent way to handle right-to-left fonts at present.]*

2.1. Testing for Unicode Capabilities

The basic text functions will be available in every Glk library. The Unicode functions may or may not be available. Before calling them, you should use the following gestalt selectors:

```
glui32 res;
res = glk_gestalt(gestalt_Unicode, 0);
```

This returns 1 if the core Unicode functions are available. If it returns 0, you should not try to call them. They may print nothing, print gibberish, or cause a run-time error. The Unicode functions include glk_buffer_to_lower_case_uni, glk_buffer_to_upper_case_uni, glk_buffer_to_title_case_uni, glk_put_char_uni, glk_put_string_uni, glk_put_buffer_uni, glk_put_char_stream_uni, glk_put_string_stream_uni, glk_put_buffer_stream_uni, glk_get_char_stream_uni, glk_get_buffer_stream_uni, glk_get_line_stream_uni, glk_request_char_event_uni, glk_request_line_event_uni, glk_stream_open_file_uni, glk_stream_open_memory_uni.

If you are writing a C program, there is an additional complication. A library which does not support Unicode may not implement the Unicode functions at all. Even if you put gestalt tests around your Unicode calls, you may get link-time errors. If the glk.h file is so old that it does not declare the Unicode functions and constants, you may even get compile-time errors.

To avoid this, you can perform a preprocessor test for the existence of GLK_MODULE_UNICODE. If this is defined, so are all the Unicode functions and constants. If not, not.

```
glui32 res;
res = glk_gestalt(gestalt_UnicodeNorm, 0);
```

This returns 1 if the Unicode normalization functions are available. If it returns 0, you should not try to call them. The Unicode normalization functions include glk_buffer_canon_decompose_uni and glk_buffer_canon_normalize_uni.

The equivalent preprocessor test for these functions is GLK_MODULE_UNICODE_NORM.

2.2. Output

When you are sending text to a window, or to a file open in text mode, you can print any of the printable Latin-1 characters: 32 to 126, 160 to 255. You can also print the newline character (linefeed, control-J, decimal 10, hex 0x0A.)

It is *not* legal to print any other control characters (0 to 9, 11 to 31, 127 to 159). You may not print even common formatting characters such as tab (control-I), carriage return (control-M), or page break (control-L). *[As usual, the behavior of the library when you print an illegal character is undefined. It is preferable that the library display a numeric code, such as "\177" or "0x7F", to warn the user that something illegal has occurred. The library may skip illegal characters entirely; but you should not rely on this.]*

Printing Unicode characters above 255 is a more complicated matter — too complicated to be covered precisely by this specification. Refer to the Unicode specification, and good luck to you.

[Unicode combining characters are a particular nuisance. Printing a combining character may alter the appearance of the previous character printed. The library should be prepared to cope with this — even if the characters are printed by two separate glk_put_char_uni() calls.]

Note that when you are sending data to a file open in binary mode, you can print any byte value, without restriction. See section 5.6.3, "File Streams".

A particular implementation of Glk may not be able to display all the printable characters. It is guaranteed to be able to display the ASCII characters (32 to 126, and the newline 10.) Other characters may be printed correctly, printed as multi-character combinations (such as "ae" for the "æ" ligature), or printed as some placeholder character (such as a bullet or question mark, or even an octal code.)

You can test for this by using the gestalt_CharOutput selector. If you set ch to a character code (Latin-1 or higher), and call

```
glui32 res, len;
res = glk_gestalt_ext(gestalt_CharOutput, ch, &len, 1);
```

then res will be one of the following values:

- gestalt_CharOutput_CannotPrint: The character cannot be meaningfully printed. If you try, the player may see nothing, or may see a placeholder.
- gestalt_CharOutput_ExactPrint: The character will be printed exactly as defined.
- gestalt_CharOutput_ApproxPrint: The library will print some approximation of the character. It will be more or less right, but it may not be precise, and it may not be distinguishable from other, similar characters. (Examples: "ae" for the "æ" ligature, "e" for "è", "|" for a broken vertical bar (¦).)

In all cases, len (the glui32 value pointed at by the third argument) will be the number of actual glyphs which will be used to represent the character. In the case of gestalt_CharOutput_ExactPrint, this will always be 1; for gestalt_CharOutput_CannotPrint, it may be 0 (nothing printed) or higher; for gestalt_CharOutput_ApproxPrint, it may be 1 or higher. This information may be useful when printing text in a fixed-width font.

[As described in section 1.9, "Other API Conventions", you may skip this information by passing NULL as the third argument in glk_gestalt_ext(), or by calling glk_gestalt() instead.]

This selector will always return gestalt_CharOutput_CannotPrint if ch is an unprintable eight-bit character (0 to 9, 11 to 31, 127 to 159.)

[Make sure you do not get confused by signed byte values. If you set a "signed char" variable ch to 0xFE, the small-thorn character (þ), it will wind up as -2. (The same is true of a "char" variable, if your compiler treats "char" as signed!) If you then call

```
res = glk_gestalt(gestalt_CharOutput, ch);
```

then (by the definition of C/C++) ch will be sign-extended *to 0xFFFFFFFE, which is* not *a legitimate character, even in Unicode. You should write*

```
res = glk_gestalt(gestalt_CharOutput, (unsigned char)ch);
```

instead.]

[Unicode includes the concept of non-spacing or combining characters, which do not represent glyphs; and double-width characters, whose glyphs take up two spaces in a fixed-width font. Future versions of this spec may recognize these concepts by returning a len of 0 or 2 when gestalt_CharOutput_ExactPrint is used. For the moment, we are adhering to a policy of "simple stuff first".]

2.3. Line Input

You can request that the player enter a line of text. See section 4.2, "Line Input Events".

This text will be placed in a buffer of your choice. There is no length field or null terminator in the buffer. (The length of the text is returned as part of the line-input event.)

If you use the basic text API, the buffer will contain only printable Latin-1 characters (32 to 126, 160 to 255).

A particular implementation of Glk may not be able to accept all Latin-1 printable characters as input. It is guaranteed to be able to accept the ASCII characters (32 to 126.)

You can test for this by using the gestalt_LineInput selector. If you set ch to a character code, and call

```
glui32 res;
res = glk_gestalt(gestalt_LineInput, ch);
```

then res will be TRUE (1) if that character can be typed by the player in line input, and FALSE (0) if not. Note that if ch is a nonprintable Latin-1 character (0 to 31, 127 to 159), then this is guaranteed to return FALSE.

2.4. Character Input

You can request that the player hit a single key. See section 4.1, "Character Input Events".

If you use the basic text API, the character code which is returned can be any value from 0 to 255. The printable character codes have already been described. The remaining codes are typically control codes: control-A to control-Z and a few others.

There are also a number of special codes, representing special keyboard keys, which can be returned from a char-input event. These are represented as 32-bit integers, starting with 4294967295 (0xFFFFFFFF) and working down. The special key codes are defined in the glk.h file. They include:

- keycode_Left, keycode_Right, keycode_Up, keycode_Down (arrow keys)
- keycode_Return (return or enter)
- keycode_Delete (delete or backspace)
- keycode_Escape
- keycode_Tab
- keycode_PageUp
- keycode_PageDown
- keycode_Home
- keycode_End
- keycode_Func1, keycode_Func2, keycode_Func3, ... keycode_Func12 (twelve function keys)
- keycode_Unknown (any key which has no Latin-1 or special code)

Various implementations of Glk will vary widely in which characters the player can enter. The most obvious limitation is that some characters are mapped to others. For example, most keyboards return a control-I code when the tab key is pressed. The Glk library, if it can recognize this at all, will generate a keycode_Tab event (value 0xFFFFFFF7) when this occurs. Therefore, for these keyboards, *no* keyboard key will generate a control-I event (value 9.) The Glk library will probably map many of the control codes to the other special keycodes.

[On the other hand, the library may be very clever and discriminate between tab and control-I. This is legal. The idea is, however, that if your program asks the player to "press the tab key", you should check for a keycode_Tab event as opposed to a control-I event.]

Some characters may not be enterable simply because they do not exist. *[Not all keyboards have a home or end key. A pen-based platform may not recognize any control characters at all.]*

Some characters may not be enterable because they are reserved for the purposes of the interface. For example, the Mac Glk library reserves the tab key for switching between different Glk windows. Therefore, on the Mac, the library will never generate a keycode_Tab event *or* a control-I event.

[Note that the linefeed or control-J character, which is the only printable *control character, is probably not* typable. *This is because, in most libraries, it will be converted to keycode_Return. Again, you should check for keycode_Return if your program asks the player to "press the return key".]*

[The delete and backspace keys are merged into a single keycode because they have such an astonishing history of being confused in the first place... this spec formally waives any desire to define the difference. Of course, a library is free to distinguish delete and backspace during line input. This is when it matters most; conflating the two during character input should not be a large problem.]

You can test for this by using the gestalt_CharInput selector. If you set ch to a character code, or a special code (from 0xFFFFFFFF down), and call

```
   glui32 res;
   res = glk_gestalt(gestalt_CharInput, ch);
```

then res will be TRUE (1) if that character can be typed by the player in character input, and FALSE (0) if not.

[Glk porters take note: it is not a goal to be able to generate every single possible key event. If the library says that it can generate a particular keycode, then game programmers will assume that it is available, and ask players to use it. *If a keycode_Home event can only be generated by typing escape-control-A, and the player does not know this, the player will be lost when the game says "Press the home key to see the next hint." It is better for the library to say that it* cannot *generate a keycode_Home event; that way the game can detect the situation and ask the user to type H instead.]*

[Of course, it is better not to rely on obscure keys in any case. The arrow keys and return are nearly certain to be available; the others are of gradually decreasing reliability, and you (the game programmer) should not depend on them. You must *be certain to check for the ones you want to use,* including *the arrow keys and return, and be prepared to use different keys in your interface if gestalt_CharInput says they are not available.]*

2.5. Upper and Lower Case

You can convert Latin-1 characters between upper and lower case with two Glk utility functions:

```
   unsigned char glk_char_to_lower(unsigned char ch);
   unsigned char glk_char_to_upper(unsigned char ch);
```

These have a few advantages over the standard ANSI tolower() and toupper() macros. They work for the entire Latin-1 character set, including accented letters; they behave consistently on all platforms, since they’re part of the Glk library; and they are safe for all characters. That is, if you call glk_char_to_lower() on a lower-case character, or a character which is not a letter, you’ll get the argument back unchanged.

The case-sensitive characters in Latin-1 are the ranges 0x41..0x5A, 0xC0..0xD6, 0xD8..0xDE (upper case) and the ranges 0x61..0x7A, 0xE0..0xF6, 0xF8..0xFE (lower case). These are arranged in parallel; so glk_char_to_lower() will add 0x20 to values in the upper-case ranges, and glk_char_to_upper() will subtract 0x20 from values in the lower-case ranges.

Unicode character conversion is trickier, and must be applied to character arrays, not single characters.

```
glui32 glk_buffer_to_lower_case_uni(glui32 *buf, glui32 len, glui32
   numchars);
glui32 glk_buffer_to_upper_case_uni(glui32 *buf, glui32 len, glui32
   numchars);
glui32 glk_buffer_to_title_case_uni(glui32 *buf, glui32 len, glui32
   numchars, glui32 lowerrest);
```

These functions provide two length arguments because a string of Unicode characters may expand when its case changes. The len argument is the available length of the buffer; numchars is the number of characters in the buffer initially. (So numchars must be less than or equal to len. The contents of the buffer after numchars do not affect the operation.)

The functions return the number of characters after conversion. If this is greater than len, the characters in the array will be safely truncated at len, but the true count will be returned. (The contents of the buffer after the returned count are undefined.)

The lower_case and upper_case functions do what you'd expect: they convert every character in the buffer (the first numchars of them) to its upper or lower-case equivalent, if there is such a thing.

The title_case function has an additional (boolean) flag. If the flag is zero, the function changes the first character of the buffer to upper-case, and leaves the rest of the buffer unchanged. If the flag is nonzero, it changes the first character to upper-case and the rest to lower-case.

See the Unicode spec (chapter 3.13, chapter 4.2, etc) for the exact definitions of upper, lower, and title-case mapping.

[Unicode has some strange case cases. For example, a combined character that looks like "ss" might properly be upper-cased into two *"S" characters. Title-casing is even stranger; "ss" (at the beginning of a word) might be title-cased into a different combined character that looks like "Ss". The glk_buffer_to_title_case_uni() function is actually title-casing the first character of the buffer. If it makes a difference.]*

[Earlier drafts of this spec had a separate function which title-cased the first character of every word *in the buffer. I took this out after reading Unicode Standard Annex #29, which explains how to divide a string into words. If you want it, feel free to implement it.]*

2.6. Unicode String Normalization

Comparing Unicode strings is difficult, because there can be several ways to represent a piece of text as a Unicode string. For example, the one-character string "è" (an accented "e") will be displayed the same as the two-character string containing "e" followed by Unicode character 0x0300 (COMBINING GRAVE ACCENT). These strings should be considered equal.

Therefore, a Glk program that accepts line input should convert its text to a normalized form before parsing it. These functions offer those conversions. The algorithms are defined by the Unicode spec (chapter 3.7) and <Unicode Standard Annex #15>.

```
glui32 glk_buffer_canon_decompose_uni(glui32 *buf, glui32 len, glui32
   numchars);
```

This transforms a string into its canonical decomposition ("Normalization Form D"). Effectively, this takes apart multipart characters into their individual parts. For example, it would convert "è" (character 0xE8, an accented "e") into the two-character string containing "e" followed by Unicode character 0x0300 (COMBINING GRAVE ACCENT). If a single character has multiple accent marks, they are also rearranged into a standard order.

```
glui32 glk_buffer_canon_normalize_uni(glui32 *buf, glui32 len, glui32
    numchars);
```

This transforms a string into its canonical decomposition and recomposition ("Normalization Form C"). Effectively, this takes apart multipart characters, and then puts them back together in a standard way. For example, this would convert the two-character string containing "e" followed by Unicode character 0x0300 (COMBINING GRAVE ACCENT) into the one-character string " è" (character 0xE8, an accented "e").

The canon_normalize function includes decomposition as part of its implementation. You never have to call both functions on the same string.

Both of these functions are idempotent.

These functions provide two length arguments because a string of Unicode characters may expand when it is transformed. The len argument is the available length of the buffer; numchars is the number of characters in the buffer initially. (So numchars must be less than or equal to len. The contents of the buffer after numchars do not affect the operation.)

The functions return the number of characters after transformation. If this is greater than len, the characters in the array will be safely truncated at len, but the true count will be returned. (The contents of the buffer after the returned count are undefined.)

[The Unicode spec also defines stronger forms of these functions, called "compatibility decomposition and recomposition" ("Normalization Form KD" and "Normalization Form KC".) These do all of the accent-mangling described above, but they also transform many other obscure Unicode characters into more familiar forms. For example, they split ligatures apart into separate letters. They also convert Unicode display variations such as script letters, circled letters, and half-width letters into their common forms.]

[The Glk spec does not currently provide these stronger transformations. Glk's expected use of Unicode normalization is for line input, and an OS facility for line input will generally not produce these alternate character forms (unless the user goes out of his way to type them). Therefore, the need for these transformations does not seem to be worth the extra data table space.]

2.6.1. A Note on Unicode Case-Folding and Normalization

With all of these Unicode transformations hovering about, an author might reasonably ask about the right way to handle line input. Our recommendation is: call glk_buffer_to_lower_case_uni(), followed by glk_buffer_canon_normalize_uni(), and then parse the result. The parsing process should of course match against strings that have been put through the same process.

The Unicode spec (chapter 3.13) gives a different, three-step process: decomposition, case-folding, and decomposition again. Our recommendation comes through a series of practical compromises:

- The initial decomposition is only necessary because of a historical error in the Unicode spec: character 0x0345 (COMBINING GREEK YPOGEGRAMMENI) behaves inconsistently. We ignore this case, and skip this step.
- Case-folding is a slightly different operation from lower-casing. (Case-folding splits some combined characters, so that, for example, "ß" can match both "ss" and "SS".) However,

Glk does not currently offer a case-folding function. We substitute glk_buffer_to_lower_case_uni().
- I'm not sure why the spec recommends decomposition (glk_buffer_canon_decompose_uni()) rather than glk_buffer_canon_normalize_uni(). However, composed characters are the norm in source code, and therefore in compiled Inform game files. If we specified decomposition, the compiler would have to do extra work; also, the standard Inform dictionary table (with its fixed word length) would store fewer useful characters. Therefore, we substitute glk_buffer_canon_normalize_uni().

[We may revisit these recommendations in future versions of the spec.]

3. Windows

On most platforms, the program/library combination will appear to the player in a window — either a window which covers the entire screen, or one which shares screen space with other windows in a multi-programming environment. Obviously your program does not have worry about the details of this. The Glk screen space is a rectangle, which you can divide into panels for various purposes. It is these panels which I will refer to as "windows" hereafter.

You refer to a window using an opaque C structure pointer. See section 1.6, "Opaque Objects".

A window has a type. Currently there are four window types:

- Text buffer windows: A stream of text. *[The "story window" of an Infocom game.]* You can only print at the end of the stream, and input a line of text at the end of the stream.
- Text grid windows: A grid of characters in a fixed-width font. *[The "status window" of an Infocom game.]* You can print anywhere in the grid.
- Graphics windows: A grid of colored pixels. Graphics windows do not support text input or output, but there are image commands to draw in them. *[This is an optional capability; not all Glk libraries support graphics. See section 7.4, "Testing for Graphics Capabilities".]*
- Blank windows: A blank window. Blank windows support neither input nor output. *[They exist mostly to be an example of a "generic" window. You are unlikely to want to use them.]*

As Glk is an expanding system, more window types may be added in the future. Therefore, it is important to remember that not all window types will necessarily be available under all Glk libraries.

There is one other special type of window, the pair window. Pair windows are created by Glk as part of the system of window arrangement. You cannot create them yourself. See section 3.5.2, "Pair Windows".

Every window has a rock. This is a value you provide when the window is created; you can use it however you want. See section 1.6.1, "Rocks".

When Glk starts up, there are no windows.

[When I say there are no windows, I mean there are no Glk windows. In a multiprogramming environment, such as X or MacOS, there may be an application window visible; this is the screen space that will contain all the Glk windows that you create. But at first, this screen space is empty and unused.]

Without a window, you cannot do any kind of input or output; so the first thing you'll want to do is create one. See section 3.2, "Window Opening, Closing, and Constraints".

You can create as many windows as you want, of any types. You control their arrangement and sizes through a fairly flexible system of calls. See section 3.1, "Window Arrangement".

You can close any windows you want. You can even close all the windows, which returns you to the original startup state.

You can request input from any or all windows. Input can be mouse input (on platforms which support a mouse), single-character input, or input of an entire line of text. It is legal to request input from several windows at the same time. The library will have some interface mechanism for the player to control which window he is typing in.

3.1. Window Arrangement

The Way of Window Arrangement is fairly complicated. I'll try to explain it coherently. *[If you are reading this document to get an overview of Glk, by all means skip forward to section 3.5, "The Types of Windows". Come back here later.]*

Originally, there are no windows. You can create a window, which will take up the entire available screen area. You can then split this window in two. One of the halves is the original window; the other half is new, and can be of any type you want. You can control whether the new window is left, right, above, or below the original one. You can also control how the split occurs. It can be 50-50, or 70-30, or any other percentage split. Or, you can give a fixed width to the new window, and allow the old one to take up the rest of the available space. Or you can give a fixed width to the *old* window, and let the *new* one take up the rest of the space.

Now you have two windows. In exactly the same way, you can split either of them — the original window, or the one you just created. Whichever one you split becomes two, which together take up the same space that the one did before.

You can repeat this as often as you want. Every time you split a window, one new window is created. Therefore, the call that does this is called glk_window_open(). *[It might have been less confusing to call it "glk_split_window" — or it might have been more confusing. I picked one.]*

It is important to remember that the order of splitting matters. If you split twice times, you don't have a trio of windows; you have a pair with another pair on one side. Mathematically, the window structure is a binary tree.

Example time. Say you do two splits, each a 50-50 percentage split. You start with the original window A, and split that into A and B; then you split B into B and C.

```
+---------+
|         |       O
|    A    |      / \
|         |     A   O
+---------+        / \
|    B    |       B   C
+---------+
|    C    |
+---------+
```

Or, you could split A into A and B, and then split A again into A and C.

```
+---------+
|    A    |       O
```

```
+---------+      / \
|    C    |     O   B
+---------+    / \
|         |   A   C
|    B    |
|         |
+---------+
```

I’m using the simplest possible splits in the examples above. Every split is 50-50, and the new window of the pair is always *below* the original one (the one that gets split.) You can get fancier than that. Here are three more ways to perform the first example; all of them have the *same* tree structure, but look different on the screen.

```
+---------+ +---------+ +---------+
|         | |    A    | |         |     O
|    A    | +---------+ |    A    |    / \
|         | |    B    | |         |   A   O
+---------+ +---------+ +----+----+      / \
|    C    | |         | |    |    |     B   C
+---------+ |    C    | | C  | B  |
|    B    | |         | |    |    |
+---------+ +---------+ +----+----+
```

On the left, we turn the second split (B into B/C) upside down; we put the new window (C) above the old window (B).

In the center, we mess with the percentages. The first split (A into A/B) is a 25-75 split, which makes B three times the size of A. The second (B into B/C) is a 33-66 split, which makes C twice the size of B. This looks rather like the second example above, but has a different internal structure.

On the right, the second split (B into B/C) is vertical instead of horizontal, with the new window (C) on the left of the old one.

The visible windows on the Glk screen are "leaf nodes" of the binary tree; they hang off the ends of the branches in the diagram. There are also the "internal nodes", the ones at the forks, which are marked as "O". These are the mysterious pair windows.

You don’t create pair windows directly; they are created as a consequence of window splits. Whenever you create a new window, a new pair window is also created automatically. In the following two-split process, you can see that when a window is split, it is replaced by a new pair window, and moves down to become one of that "O"’s two children.

```
+---+    A
|   |
| A |
|   |
+---+

+---+    O
| A |   / \
+---+  A   B
| B |
+---+
+---+    O
| A |   / \
+-+-+  A   O
|C|B|     / \
+-+-+    B   C
```

You can’t draw into a pair window. It’s completely filled up with the two windows it contains. They’re what you should be drawing into.

Why have pair windows in the system at all? They’re convenient for certain operations. For example, you can close any window at any time; but sometimes you want to close an entire nest of windows at once. In the third stage shown, if you close the lower pair window, it blows away all its descendents — both B and C — and leaves just a single window, A, which is what you started with.

I’m using some math terminology already, so I’ll explain it briefly. The "root" of the tree is the top (math trees, like family trees, grow upside down.) If there’s only one window, it’s the root; otherwise the root is the topmost "O". Every pair window has exactly two "children". Other kinds of windows are leaves on the tree, and have no children. A window’s "descendants", obviously, are its children and grandchildren and great-grandchildren and so on. The "parent" and "ancestors" of a window are exactly what you’d expect. So the root window is the ancestor of every other window.

There are Glk functions to determine the root window, and to determine the parent of any given window. Note that every window’s parent is a pair window. (Except for the root window, which has no parent.)

3.2. Window Opening, Closing, and Constraints

```
winid_t glk_window_open(winid_t split, glui32 method, glui32 size, glui32
   wintype, glui32 rock);
```

If there are no windows, the first three arguments are meaningless. split *must* be zero, and method and size are ignored. wintype is the type of window you’re creating, and rock is the rock (see section 1.6.1, "Rocks").

If any windows exist, new windows must be created by splitting existing ones. split is the window you want to split; this *must not* be zero. method specifies the direction and the split method (see below). size is the size of the split. wintype is the type of window you’re creating, and rock is the rock.

The winmethod constants:

- winmethod_Above, winmethod_Below, winmethod_Left, winmethod_Right: The new window will be above, below, to the left, or to the right of the old one which was split.
- winmethod_Fixed, winmethod_Proportional: The new window is a fixed size, or a given proportion of the old window’s size. (See below.)
- winmethod_Border, winmethod_NoBorder: There should or should not be a visible window border between the new window and its sibling. (This is a hint to the library; you might specify NoBorder between two graphics windows that should form a single image.)

The method argument must be the logical-or of a direction constant (winmethod_Above, winmethod_Below, winmethod_Left, winmethod_Right) and a split-method constant (winmethod_Fixed, winmethod_Proportional).

Remember that it is possible that the library will be unable to create a new window, in which case glk_window_open() will return NULL. *[It is acceptable to gracefully exit, if the window you are creating is an important one — such as your first window. But you should not try to perform any window operation on the id until you have tested to make sure it is non-zero.]*

The examples we've seen so far have the simplest kind of size control. (Yes, this is "below".) Every pair is a percentage split, with X percent going to one side, and (100-X) percent going to the other side. If the player resizes the window, the whole mess expands, contracts, or stretches in a uniform way.

As I said above, you can also make fixed-size splits. This is a little more complicated, because you have to know how this fixed size is measured.

Sizes are measured in a way which is different for each window type. For example, a text grid window is measured by the size of its fixed-width font. You can make a text grid window which is fixed at a height of four rows, or ten columns. A text buffer window is measured by the size of *its* font. *[Remember that different windows may use different size fonts. Even two text grid windows may use fixed-size fonts of different sizes.]* Graphics windows are measured in pixels, not characters. Blank windows aren't measured at all; there's no meaningful way to measure them, and therefore you can't create a blank window of a fixed size, only of a proportional (percentage) size.

So to create a text buffer window which takes the top 40% of the original window's space, you would execute

```
newwin = glk_window_open(win, winmethod_Above | winmethod_Proportional, 40,
   wintype_TextBuffer, 0);
```

To create a text grid which is always five lines high, at the bottom of the original window, you would do

```
newwin = glk_window_open(win, winmethod_Below | winmethod_Fixed, 5,
   wintype_TextGrid, 0);
```

Note that the meaning of the size argument depends on the method argument. If the method is winmethod_Fixed, it also depends on the wintype argument. The new window is then called the "key window" of this split, because its window type determines how the split size is computed. *[For winmethod_Proportional splits, you can still call the new window the "key window". But the key window is not important for proportional splits, because the size will always be computed as a simple ratio of the available space, not a fixed size of one child window.]*

This system is more or less peachy as long as all the constraints work out. What happens when there is a conflict? The rules are simple. Size control always flows down the tree, and the player is at the top. Let's bring out an example:

```
+---------+
| C: 2    |
|    rows |       O
+---------+      / \
| A       |     O   B
+---------+    / \
|         |   A   C
```

```
   | B: 50%  |
   |         |
   |         |
   +---------+
```

First we split A into A and B, with a 50% proportional split. Then we split A into A and C, with C above, C being a text grid window, and C gets a fixed size of two rows (as measured in its own font size). A gets whatever remains of the 50% it had before.

Now the player stretches the window vertically.

```
   +---------+
   | C: 2    |
   |    rows |
   +---------+
   | A       |
   |         |
   +---------+
   |         |
   |         |
   | B: 50%  |
   |         |
   |         |
   +---------+
```

The library figures: the topmost split, the original A/B split, is 50-50. So B gets half the screen space, and the pair window next to it (the lower "O") gets the other half. Then it looks at the lower "O". C gets two rows; A gets the rest. All done.

Then the user maliciously starts squeezing the window down, in stages:

```
   +---------+  +---------+  +---------+  +---------+  +---------+
   | C: 2    |  |    C    |  |    C    |  |    C    |  +---------+
   |    rows |  |         |  |         |  +---------+  +---------+
   +---------+  +---------+  +---------+  +---------+  |    B    |
   | A       |  |    A    |  +---------+  |    B    |  +---------+
   |         |  +---------+  |         |  |         |
   +---------+  |         |  |    B    |  +---------+
   |         |  |    B    |  |         |
   |         |  |         |  +---------+
   | B: 50%  |  |         |
   |         |  +---------+
   |         |
   +---------+
```

The logic remains the same. B always gets half the space. At stage 3, there’s no room left for A, so it winds up with zero height. Nothing displayed in A will be visible. At stage 4, there isn’t even room in the upper 50% to give C its two rows; so it only gets one. Finally, C is squashed out of existence as well.

When a window winds up undersized, it remembers what size it should be. In the example above, A remembers that it should be two rows; if the user expands the window to the original size, it would return to the original layout.

The downward flow of control is a bit harsh. After all, in stage 4, there's room for C to have its two rows if only B would give up some of its 50%. But this does not happen. *[This makes life much easier for the Glk library. To determine the configuration of a window, it only needs to look at the window's ancestors, never at its descendants. So window layout is a simple recursive algorithm, no backtracking.]*

What happens when you split a fixed-size window? The resulting pair window — that is, the two new parts together — retain the same size constraint as the original window that was split. The key window for the original split is still the key window for that split, even though it's now a grandchild instead of a child.

The easy, and correct, way to think about this is that the size constraint is stored by a window's parent, not the window itself; and a constraint consists of a pointer to a key window plus a size value.

```
+---------+       +---------+         +---------+
|         |       |         |         | C: 2    |
|         |  A    | A: 50%  |   O1    |    rows |     O1
|         |       |         |  / \    +---------+    / \
|         |       |         | A   B   | A       |   O2  B
| A       |       +---------+         +---------+  / \
|         |       |         |         |         | A   C
|         |       | B       |         | B       |
|         |       |         |         |         |
|         |       |         |         |         |
+---------+       +---------+         +---------+
```

The initial window is A. After the first split, the new pair window (O1, which covers the whole screen) knows that its new child (B) is below A, and gets 50% of its own area. (B is the key window for this split, but a proportional split doesn't care about key windows.)

After the *second* split, all this remains true; O1 knows that its first child gets 50% of its space, and B is O1's key window. But now O1's first child is O2 instead of A. The newer pair window (O2) knows that *its* first child (C) is above the second, and gets a fixed size of two rows. (As measured in C's font, because C is O2's key window.)

If we split C, now, the resulting pair will still be two C-font rows high — that is, tall enough for two lines of whatever font C displays. For the sake of example, we'll do this vertically.

```
+----+----+
| C  | D  |
|    |    |     O1
+----+----+    / \
| A       |   O2  B
+---------+  / \
|         | A   O3
| B       |    / \
|         |   C   D
|         |
+---------+
```

O3 now knows that its children have a 50-50 left-right split. O2 is still committed to giving its upper child, O3, two C-font rows. Again, this is because C is O2's key window. *[This turns out to be a good idea, because it means that C, the text grid window, is still two rows high. If O3 had been a upper-lower split, things wouldn't work out so neatly. But the rules would still apply. If you don't like this, don't do it.]*

```
void glk_window_close(winid_t win, stream_result_t *result);
```

This closes a window, which is pretty much exactly the opposite of opening a window. It is legal to close all your windows, or to close the root window (which does the same thing.)

The result argument is filled with the output character count of the window stream. See section 5, "Streams" and section 5.3, "Closing Streams".

When you close a window (and it is not the root window), the other window in its pair takes over all the freed-up area. Let's close D, in the current example:

```
+---------+
| C       |
|         |     O1
+---------+    / \
| A       |   O2  B
+---------+  / \
|         | A   C
| B       |
|         |
|         |
+---------+
```

Notice what has happened. D is gone. O3 is gone, and its 50-50 left-right split has gone with it. The other size constraints are unchanged; O2 is still committed to giving its upper child two rows, as measured in the font of O2's key window, which is C. Conveniently, O2's upper child *is* C, just as it was before we created D. In fact, now that D is gone, everything is back to the way it was before we created D.

But what if we had closed C instead of D? We would have gotten this:

```
+---------+
+---------+
|         |     O1
| A       |    / \
|         |   O2  B
+---------+  / \
|         | A   D
| B       |
|         |
|         |
+---------+
```

Again, O3 is gone. But D has collapsed to zero height. This is because its height is controlled by O2, and O2's key window was C, and C is now gone. O2 no longer has a key window at all, so it cannot compute a height for its upper child, so it defaults to zero.

[This may seem to be an inconvenient choice. That is deliberate. You should not leave a pair window with no key, and the zero-height default reminds you not to. You can use glk_window_set_arrangement() to set a new split measurement and key window. See section 3.3, "Changing Window Constraints".]

3.3. Changing Window Constraints

There are library functions to change and to measure the size of a window.

```
void glk_window_get_size(winid_t win, glui32 *widthptr, glui32 *heightptr);
void glk_window_set_arrangement(winid_t win, glui32 method, glui32 size,
   winid_t keywin);
void glk_window_get_arrangement(winid_t win, glui32 *methodptr, glui32
   *sizeptr, winid_t *keywinptr);
```

glk_window_get_size() simply returns the actual size of the window, in its measurement system. As described in section 1.9, "Other API Conventions", either widthptr or heightptr can be NULL, if you only want one measurement. *[Or, in fact, both, if you want to waste time.]*

glk_window_set_arrangement() changes the size of an existing split — that is, it changes the constraint of a given pair window. glk_window_get_arrangement() returns the constraint of a given pair window.

Consider the example above, where D has collapsed to zero height. Say D was a text buffer window. You could make a more useful layout by doing

```
winid_t o2;
o2 = glk_window_get_parent(d);
glk_window_set_arrangement(o2, winmethod_Above | winmethod_Fixed, 3, d);
```

That would set the D (the upper child of O2) to be O2's key window, and give it a fixed size of 3 rows.

If you later wanted to expand D, you could do

```
glk_window_set_arrangement(o2, winmethod_Above | winmethod_Fixed, 5, NULL);
```

That expands D to five rows. Note that, since O2's key window is already set to D, it is not necessary to provide the keywin argument; you can pass NULL to mean "leave the key window unchanged."

If you do change the key window of a pair window, the new key window *must* be a descendant of that pair window. In the current example, you could change O2's key window to be A, but not B. The key window also cannot be a pair window itself.

```
glk_window_set_arrangement(o2, winmethod_Below | winmethod_Fixed, 3, NULL);
```

This changes the constraint to be on the *lower* child of O2, which is A. The key window is still D; so A would then be three rows high as measured in D's font, and D would get the rest of O2's space. That may not be what you want. To set A to be three rows high as measured in A's font, you would do

```
glk_window_set_arrangement(o2, winmethod_Below | winmethod_Fixed, 3, a);
```

Or you could change O2 to a proportional split:

```
    glk_window_set_arrangement(o2, winmethod_Below | winmethod_Proportional,
       30, NULL);
```

or

```
    glk_window_set_arrangement(o2, winmethod_Above | winmethod_Proportional,
       70, NULL);
```

These do exactly the same thing, since 30% above is the same as 70% below. You don't need to specify a key window with a proportional split, so the keywin argument is NULL. (You could actually specify either A or D as the key window, but it wouldn't affect the result.)

Whatever constraint you set, glk_window_get_size() will tell you the actual window size you got.

Note that you can resize windows, and alter the Border/NoBorder flag. But you can't flip or rotate them. You can't move A above D, or change O2 to a vertical split where A is left or right of D. *[To get this effect you could close one of the windows, and re-split the other one with glk_window_open().]*

3.4. A Note on Display Style

The way windows are displayed is, of course, entirely up to the Glk library; it depends on what is natural for the player's machine. The borders between windows may be black lines, 3-D bars, rows of "#" characters; there may even be no borders at all. The library may not support the Border/NoBorder hint, in which case *every* pair of windows will have a visible border — or no border — between them.

[The Border/NoBorder was introduced in Glk 0.7.1. Prior to that, all games used the Border hint, and this remains the default. However, as noted, not all implementations display window borders. Therefore, for existing implementations, "Border" may be understood as "your normal style of window display"; "NoBorder" may be understood as "suppress any interwindow borders you may have".]

There may be decorations within the windows as well. A text buffer window will often have a scroll bar. The library (or player) may prefer wide margins around each text window. And so on.

The library is reponsible for handling these decorations, margins, spaces, and borders. You should never worry about them. You are guaranteed that if you request a fixed size of two rows, your text grid window will have room for two rows of characters — if there is enough total space. Any margins or borders will be allowed for already. If there *isn't* enough total space (as in stages 4 and 5, above), you lose, of course.

How do you know when you're losing? You can call glk_window_get_size() to determine the window size you really got. Obviously, you should draw into your windows based on their real size, not the size you requested. If there's enough space, the requested size and the real size will be identical; but you should not rely on this. Call glk_window_get_size() and check.

3.5. The Types of Windows

This is a technical description of all the window types, and exactly how they behave.

3.5.1. Blank Windows

A blank window is always blank. It supports no input and no output. (You can call glk_window_get_stream() on it, as you can with any window, but printing to the resulting stream has no effect.) A blank window has no size; glk_window_get_size() will return (0,0), and it is illegal to set a window split with a fixed size in the measurement system of a blank window.

[A blank window is not the same as there being no windows. When Glk starts up, there are no windows at all, not even a window of the blank type.]

3.5.2. Pair Windows

A pair window is completely filled by the two windows it contains. It supports no input and no output, and it has no size.

You cannot directly create a pair window; one is automatically created every time you split a window with glk_window_open(). Pair windows are always created with a rock value of 0.

You can close a pair window with glk_window_close(); this also closes every window contained within the pair window.

It is legal to split a pair window when you call glk_window_open().

3.5.3. Text Buffer Windows

A text buffer window contains a linear stream of text. It supports output; when you print to it, the new text is added to the end. There is no way for you to affect text which has already been printed. There are no guarantees about how much text the window keeps; old text may be stored forever, so that the user can scroll back to it, or it may be thrown away as soon as it scrolls out of the window. *[Therefore, there may or may not be a player-controllable scroll bar or other scrolling widget.]*

The display of the text in a text buffer is up to the library. Lines will probably not be broken in the middles of words — but if they are, the library is not doing anything illegal, only ugly. Text selection and copying to a clipboard, if available, are handled however is best on the player’s machine. Paragraphs (as defined by newline characters in the output) may be indented. *[You should not, in general, fake this by printing spaces before each paragraph of prose text. Let the library and player preferences handle that. Special cases (like indented lists) are of course up to you.]*

When a text buffer is cleared (with glk_window_clear()), the library will do something appropriate; the details may vary. It may clear the window, with later text appearing at the top — or the bottom. It may simply print enough blank lines to scroll the current text out of the window. It may display a distinctive page-break symbol or divider.

The size of a text buffer window is necessarily imprecise. Calling glk_window_get_size() will return the number of rows and columns that would be available *if* the window was filled with "0" (zero) characters in the "normal" font. However, the window may use a non-fixed-width font, so that number of characters in a line could vary. The window might even support variable-height text (say, if the player is using large text for emphasis); that would make the number of lines in the window vary as well.

Similarly, when you set a fixed-size split in the measurement system of a text buffer, you are setting a window which can handle a fixed number of rows (or columns) of "0" characters. The number of rows (or characters) that will actually be displayed depends on font variances.

A text buffer window supports both character and line input, but not mouse input.

In character input, there will be some visible signal that the window is waiting for a keystroke. (Typically, a cursor at the end of the text.) When the player hits a key in that window, an event is generated, but the key is *not* printed in the window.

In line input, again, there will be some visible signal. It is most common for the player to compose input in the window itself, at the end of the text. (This is how IF story input usually looks.) But it's not strictly required. An alternative approach is the way MUD clients usually work: there is a dedicated one-line input window, outside of Glk's window space, and the user composes input there. *[If this approach is used, there will still be some way to handle input from two windows at once. It is the library's responsibility to make this available to the player. You only need request line input and wait for the result.]*

By default, when the player finishes his line of input, the library will display the input text at the end of the buffer text (if it wasn't there already.) It will be followed by a newline, so that the next text you print will start a new line (paragraph) after the input.

If you call glk_cancel_line_event(), the same thing happens; whatever text the user was composing is visible at the end of the buffer text, followed by a newline.

However, this default behavior can be changed with the glk_set_echo_line_event() call. If the default echoing is disabled, the library will *not* display the input text (plus newline) after input is either completed or cancelled. The buffer will end with whatever prompt you displayed before requesting input. If you want the traditional input behavior, it is then your responsibility to print the text, using the Input text style, followed by a newline (in the original style).

3.5.4. Text Grid Windows

A text grid contains a rectangular array of characters, in a fixed-width font. Its size is the number of columns and rows of the array.

A text grid window supports output. It maintains knowledge of an output cursor position. When the window is opened, it is filled with blanks (space characters), and the output cursor starts in the top left corner — character (0,0). If the window is cleared with glk_window_clear(), the window is filled with blanks again, and the cursor returns to the top left corner.

When you print, the characters of the output are laid into the array in order, left to right and top to bottom. When the cursor reaches the end of a line, or if a newline (0x0A) is printed, the cursor goes to the beginning of the next line. The library makes *no* attempt to wrap lines at word breaks. If the cursor reaches the end of the last line, further printing has no effect on the window until the cursor is moved.

[Note that printing fancy characters may cause the cursor to advance more than one position per character. (For example, the "æ" ligature may print as two characters.) See section 2.2, "Output", for how to test this situation.]

You can set the cursor position with glk_window_move_cursor().

```
   void glk_window_move_cursor(winid_t win, glui32 xpos, glui32 ypos);
```

If you move the cursor right past the end of a line, it wraps; the next character which is printed will appear at the beginning of the next line.

If you move the cursor below the last line, or when the cursor reaches the end of the last line, it goes "off the screen" and further output has no effect. You must call glk_window_move_cursor() or glk_window_clear() to move the cursor back into the visible region.

[Note that the arguments of glk_window_move_cursor are unsigned ints. This is okay, since there are no negative positions. If you try to pass a negative value, Glk will interpret it as a huge positive value, and it will wrap or go off the last line.]

[Also note that the output cursor is not *necessarily visible. In particular, when you are requesting line or character input in a grid window, you cannot rely on the cursor position to prompt the player where input is indicated. You should print some character prompt at that spot — a ">" character, for example.]*

When a text grid window is resized smaller, the bottom or right area is thrown away, but the remaining area stays unchanged. When it is resized larger, the new bottom or right area is filled with blanks. *[You may wish to watch for evtype_Arrange events, and clear-and-redraw your text grid windows when you see them change size.]*

Text grid window support character and line input, as well as mouse input (if a mouse is available.)

Mouse input returns the position of the character that was touched, from (0,0) to (width-1,height-1).

Character input is as described in the previous section.

Line input is slightly different; it is guaranteed to take place in the window, at the output cursor position. The player can compose input only to the right edge of the window; therefore, the maximum input length is (windowwidth - 1 - cursorposition). If the maxlen argument of glk_request_line_event() is smaller than this, the library will not allow the input cursor to go more than maxlen characters past its start point. *[This allows you to enter text in a fixed-width field, without the player being able to overwrite other parts of the window.]*

When the player finishes his line of input, it will remain visible in the window, and the output cursor will be positioned at the beginning of the *next* row. Again, if you glk_cancel_line_event(), the same thing happens. The glk_set_echo_line_event() call has no effect in grid windows.

3.5.5. Graphics Windows

A graphics window contains a rectangular array of pixels. Its size is the number of columns and rows of the array.

Each graphics window has a background color, which is initially white. You can change this; see section 7.2, "Graphics in Graphics Windows".

When a graphics window is resized smaller, the bottom or right area is thrown away, but the remaining area stays unchanged. When it is resized larger, the new bottom or right area is filled with the background color. *[You may wish to watch for evtype_Arrange events, and clear-and-redraw your graphics windows when you see them change size.]*

In some libraries, you can receive a graphics-redraw event (evtype_Redraw) at any time. This signifies that the window in question has been cleared to its background color, and must be redrawn. If you create any graphics windows, you *must* handle these events.

[Redraw events can be triggered when a Glk window is uncovered or made visible by the platform's window manager. On the other hand, some Glk libraries handle these problem automatically — for example, with a backing store — and do not send you redraw events. On the third hand, the backing store may be discarded if memory is low, or for other reasons — perhaps the screen's color depth has changed. So redraw events are always a

possibility, even in clever libraries. This is why you must be prepared to handle them.]

[However, you will not receive a redraw event when you create a graphics window. It is assumed that you will do the initial drawing of your own accord. You also do not get redraw events when a graphics window is enlarged. If you ordered the enlargement, you already know about it; if the player is responsible, you receive a window-arrangement event, which covers the situation.]

For a description of the drawing functions that apply to graphics windows, see section 7.2, "Graphics in Graphics Windows".

Graphics windows support no text input or output.

Not all libraries support graphics windows. You can test whether Glk graphics are available using the gestalt system. In a C program, you can also test whether the graphics functions are defined at compile-time. See section 7.4, "Testing for Graphics Capabilities". *[As with all windows, you should also test for NULL when you create a graphics window.]*

3.6. Echo Streams

Every window has an associated window stream; you print to the window by printing to this stream. However, it is possible to attach a second stream to a window. Any text printed to the window is also echoed to this second stream, which is called the window's "echo stream."

Effectively, any call to glk_put_char() (or the other output commands) which is directed to the window's window stream, is replicated to the window's echo stream. This also goes for the style commands such as glk_set_style().

Note that the echoing is one-way. You can still print text directly to the echo stream, and it will go wherever the stream is bound, but it does not back up and appear in the window.

```
void glk_window_set_echo_stream(winid_t win, strid_t str);
strid_t glk_window_get_echo_stream(winid_t win);
```

Initially, a window has no echo stream, so glk_window_get_echo_stream(win) will return NULL. You can set a window's echo stream to be any valid output stream by calling glk_window_set_echo_stream(win, str). You can reset a window to stop echoing by calling glk_window_set_echo_stream(win, NULL).

An echo stream can be of any type, even another window's window stream. *[This would be somewhat silly, since it would mean that any text printed to the window would be duplicated in another window. More commonly, you would set a window's echo stream to be a file stream, in order to create a transcript file from that window.]*

A window can only have one echo stream. But a single stream can be the echo stream of any number of windows, sequentially or simultaneously.

If a window is closed, its echo stream remains open; it is *not* automatically closed. *[Do not confuse the window's window stream with its echo stream. The window stream is "owned" by the window, and dies with it. The echo stream is merely temporarily associated with the window.]*

If a stream is closed, and it is the echo stream of one or more windows, those windows are reset to not echo anymore. (So then calling glk_window_get_echo_stream() on them will return NULL.)

It is illegal to set a window's echo stream to be its *own* window stream. That would create an infinite loop, and is nearly certain to crash the Glk library. It is similarly illegal to create a longer loop (two or more windows echoing to each other.)

3.7. Other Window Functions

```
    winid_t glk_window_iterate(winid_t win, glui32 *rockptr);
```

This function can be used to iterate through the list of all open windows (including pair windows.) See section 1.6.2, "Iterating Through Opaque Objects".

As that section describes, the order in which windows are returned is arbitrary. The root window is not necessarily first, nor is it necessarily last.

```
    glui32 glk_window_get_rock(winid_t win);
```

This returns the window’s rock value. Pair windows always have rock 0; all other windows return whatever rock you created them with.

```
    glui32 glk_window_get_type(winid_t win);
```

This returns the window’s type (wintype_...)

```
    winid_t glk_window_get_parent(winid_t win);
```

This returns the window which is the parent of the given window. If win is the root window, this returns NULL, since the root window has no parent. Remember that the parent of every window is a pair window; other window types are always childless.

```
    winid_t glk_window_get_sibling(winid_t win);
```

This returns the other child of the given window’s parent. If win is the root window, this returns NULL.

```
    winid_t glk_window_get_root(void);
```

This returns the root window. If there are no windows, this returns NULL.

```
    void glk_window_clear(winid_t win);
```

Erase the window. The meaning of this depends on the window type.

- Text buffer: This may do any number of things, such as delete all text in the window, or print enough blank lines to scroll all text beyond visibility, or insert a page-break marker which is treated specially by the display part of the library.
- Text grid: This will clear the window, filling all positions with blanks (in the normal style). The window cursor is moved to the top left corner (position 0,0).
- Graphics: Clears the entire window to its current background color. See section 3.5.5, "Graphics Windows".
- Other window types: No effect.

It is illegal to erase a window which has line input pending.

```
    strid_t glk_window_get_stream(winid_t win);
```

This returns the stream which is associated with the window. (See section 5.6.1, "Window Streams".) Every window has a stream which can be printed to, but this may not be useful, depending on the window type. *[For example, printing to a blank window's stream has no effect.]*

```
   void glk_set_window(winid_t win);
```

This sets the current stream to the window's stream. If win is NULL, it is equivalent to

```
   glk_stream_set_current(NULL);
```

If win is not NULL, it is equivalent to

```
   glk_stream_set_current(glk_window_get_stream(win));
```

See section 5, "Streams".

4. Events

As described above, all player input is handed to your program by the glk_select() call, in the form of events. You should write at least one event loop to retrieve these events.

```
   void glk_select(event_t *event);


   typedef struct event_struct {
       glui32 type;
       winid_t win;
       glui32 val1, val2;
   } event_t;
```

This causes the program to wait for an event, and then store it in the structure pointed to by the argument. Unlike most Glk functions that take pointers, the argument of glk_select() may not be NULL.

Most of the time, you only get the events that you request. However, there are some events which can arrive at any time. This is why you must always call glk_select() in a loop, and continue the loop until you get the event you really want.

The event structure is self-explanatory. type is the event type. The window that spawned the event, if relevant, is in win. The remaining fields contain more information specific to the event.

The event types are:

- evtype_None: No event. This is a placeholder, and glk_select() never returns it.
- evtype_Timer: An event that repeats at fixed intervals.
- evtype_CharInput: A keystroke event in a window.
- evtype_LineInput: A full line of input completed in a window.
- evtype_MouseInput: A mouse click in a window.
- evtype_Arrange: An event signalling that the sizes of some windows have changed.
- evtype_Redraw: An event signalling that graphics windows must be redrawn.
- evtype_SoundNotify: The completion of a sound being played in a sound channel.
- evtype_Hyperlink: The selection of a hyperlink in a window.

- evtype_VolumeNotify: The completion of a gradual volume change in a sound channel.

Note that evtype_None is zero, and the other values are positive. Negative event types (0x80000000 to 0xFFFFFFFF) are reserved for implementation-defined events.

You can also inquire if an event is available, without stopping to wait for one to occur.

```
void glk_select_poll(event_t *event);
```

This checks if an internally-spawned event is available. If so, it stores it in the structure pointed to by event. If not, it sets event->type to evtype_None. Either way, it returns almost immediately.

The first question you now ask is, what is an internally-spawned event? glk_select_poll() does *not* check for or return evtype_CharInput, evtype_LineInput, or evtype_MouseInput events. It is intended for you to test conditions which may have occurred while you are computing, and not interfacing with the player. For example, time may pass during slow computations; you can use glk_select_poll() to see if a evtype_Timer event has occured. (See section 4.4, "Timer Events".)

At the moment, glk_select_poll() checks for evtype_Timer, and possibly evtype_Arrange and evtype_SoundNotify events. But see section 4.9, "Other Events".

The second question is, what does it mean that glk_select_poll() returns "almost immediately"? In some Glk libraries, text that you send to a window is buffered; it does not actually appear until you request player input with glk_select(). glk_select_poll() attends to this buffer-flushing task in the same way. (Although it does not do the "Hit any key to scroll down" waiting which may be done in glk_select(); that’s a player-input task.)

Similarly, on multitasking platforms, glk_select() may yield time to other processes; and glk_select_poll() does this as well.

The upshot of this is that you should not call glk_select_poll() very often. If you are not doing much work between player inputs, you should not need to call it at all. *[For example, in a virtual machine interpreter, you should not call glk_select_poll() after every opcode.]* However, if you are doing intense computation, you may wish to call glk_select_poll() every so often to yield time to other processes. And if you are printing intermediate results during this computation, you should glk_select_poll() every so often, so that you can be certain your output will be displayed before the next glk_select().

[However, you should call glk_tick() often — once per opcode in a VM interpreter. See section 1.4, "The Tick Thing".]

4.1. Character Input Events

You can request character input from text buffer and text grid windows. There are separate functions for requesting Latin-1 input and Unicode input; see section 2.1, "Testing for Unicode Capabilities".

```
void glk_request_char_event(winid_t win);
```

Request input of a Latin-1 character or special key. A window cannot have requests for both character and line input at the same time. Nor can it have requests for character input of both types (Latin-1 and Unicode). It is illegal to call glk_request_char_event() if the window already has a pending request for either character or line input.

```
   void glk_request_char_event_uni(winid_t win);
```

Request input of a Unicode character or special key.

```
   void glk_cancel_char_event(winid_t win);
```

This cancels a pending request for character input. (Either Latin-1 or Unicode.) For convenience, it is legal to call glk_cancel_char_event() even if there is no character input request on that window. Glk will ignore the call in this case.

If a window has a pending request for character input, and the player hits a key in that window, glk_select() will return an event whose type is evtype_CharInput. Once this happens, the request is complete; it is no longer pending. You must call glk_request_char_event() or glk_request_char_event_uni() if you want another character from that window.

In the event structure, win tells what window the event came from. val1 tells what character was entered; this will be a character code, or a special keycode. (See section 2.4, "Character Input".) If you called glk_request_char_event(), val1 will be in 0..255, or else a special keycode. In any case, val2 will be 0.

4.2. Line Input Events

You can request line input from text buffer and text grid windows. There are separate functions for requesting Latin-1 input and Unicode input; see section 2.1, "Testing for Unicode Capabilities".

```
   void glk_request_line_event(winid_t win, char *buf, glui32 maxlen, glui32
      initlen);
```

Request input of a line of Latin-1 characters. A window cannot have requests for both character and line input at the same time. Nor can it have requests for line input of both types (Latin-1 and Unicode). It is illegal to call glk_request_line_event() if the window already has a pending request for either character or line input.

The buf argument is a pointer to space where the line input will be stored. (This may not be NULL.) maxlen is the length of this space, in bytes; the library will not accept more characters than this. If initlen is nonzero, then the first initlen bytes of buf will be entered as pre-existing input — just as if the player had typed them himself. *[The player can continue composing after this pre-entered input, or delete it or edit as usual.]*

The contents of the buffer are undefined until the input is completed (either by a line input event, or glk_cancel_line_event(). The library may or may not fill in the buffer as the player composes, while the input is still pending; it is illegal to change the contents of the buffer yourself.

```
   void glk_request_line_event_uni(winid_t win, glui32 *buf, glui32 maxlen,
      glui32 initlen);
```

Request input of a line of Unicode characters. This works the same as glk_request_line_event(), except the result is stored in an array of glui32 values instead of an array of characters, and the values may be any valid Unicode code points.

If possible, the library should return fully composed Unicode characters, rather than strings of base and composition characters.

[Fully-composed characters are the norm for Unicode text, so an implementation that ignores this issue will

probably produce the right result. However, the game may not want to rely on that. Another factor is that case-folding can (occasionally) produce non-normalized text. Therefore, to cover all its bases, a game should call glk_buffer_to_lower_case_uni(), followed by glk_buffer_canon_normalize_uni(), before parsing.]

[Earlier versions of this spec said that line input must always be in Unicode Normalization Form C. However, this has not been universally implemented. It is also somewhat redundant, for the results noted above. Therefore, we now merely recommend that line input be fully composed. The game is ultimately responsible for all case-folding and normalization. See section 2.6, "Unicode String Normalization".]

```
   void glk_cancel_line_event(winid_t win, event_t *event);
```

This cancels a pending request for line input. (Either Latin-1 or Unicode.) The event pointed to by the event argument will be filled in as if the player had hit enter, and the input composed so far will be stored in the buffer; see below. If you do not care about this information, pass NULL as the event argument. (The buffer will still be filled.)

For convenience, it is legal to call glk_cancel_line_event() even if there is no line input request on that window. The event type will be set to evtype_None in this case.

```
   void glk_set_echo_line_event(winid_t win, glui32 val);
```

Normally, after line input is completed or cancelled in a buffer window, the library ensures that the complete input line (or its latest state, after cancelling) is displayed at the end of the buffer, followed by a newline. This call allows you to suppress this behavior. If the val argument is zero, all *subsequent* line input requests in the given window will leave the buffer unchanged after the input is completed or cancelled; the player's input will not be printed. If val is nonzero, subsequent input requests will have the normal printing behavior.

[Note that this feature is unrelated to the window's echo stream.]

```
   res = glk_gestalt(gestalt_LineInputEcho, 0);
```

Not all libraries support this feature. This returns 1 if glk_set_echo_line_event() is supported, and 0 if it is not. *[Remember that if it is not supported, the behavior is always the default, which is line echoing* enabled.*]*

If you turn off line input echoing, you can reproduce the standard input behavior by following each line input event (or line input cancellation) by printing the input line, in the Input style, followed by a newline in the original style.

The glk_set_echo_line_event() does not affect a pending line input request. It also has no effect in non-buffer windows. *[In a grid window, the game can overwrite the input area at will, so there is no need for this distinction.]*

```
   void glk_set_terminators_line_event(winid_t win, glui32 *keycodes, glui32
      count);
```

If a window has a pending request for line input, the player can generally hit the enter key (in that window) to complete line input. The details will depend on the platform's native user interface.

It is possible to request that other keystrokes complete line input as well. (This allows a game to intercept function keys or other special keys during line input.) To do this, call glk_set_terminators_line_event(), and pass an array of count keycodes. These must all be special keycodes (see section 2.4, "Character Input"). Do not include regular printable characters in the array, nor keycode_Return (which represents the default enter key and will always be recognized). To return to the default behavior, pass a NULL or empty array.

The glk_set_terminators_line_event() affects *subsequent* line input requests in the given window. It does not affect a pending line input request. *[This distinction makes life easier for interpreters that set up UI callbacks only at the start of input.]*

A library may not support this feature; if it does, it may not support all special keys as terminators. (Some keystrokes are reserved for OS or interpreter control.)

```
   res = glk_gestalt(gestalt_LineTerminators, 0);
```

This returns 1 if glk_set_terminators_line_event() is supported, and 0 if it is not.

```
   res = glk_gestalt(gestalt_LineTerminatorKey, ch);
```

This returns 1 if the keycode ch can be passed to glk_set_terminators_line_event(). If it returns 0, that keycode will be ignored as a line terminator. Printable characters and keycode_Return will always return 0.

When line input is completed, glk_select() will return an event whose type is evtype_LineInput. Once this happens, the request is complete; it is no longer pending. You must call glk_request_line_event() if you want another line of text from that window.

In the event structure, win tells what window the event came from. val1 tells how many characters were entered. val2 will be 0 unless input was ended by a special terminator key, in which case val2 will be the keycode (one of the values passed to glk_set_terminators_line_event()).

The characters themselves are stored in the buffer specified in the original glk_request_line_event() or glk_request_line_event_uni() call. *[There is no null terminator or newline stored in the buffer.]*

It is illegal to print anything to a window which has line input pending. *[This is because the window may be displaying and editing the player's input, and printing anything would make life unnecessarily complicated for the library.]*

4.3. Mouse Input Events

On some platforms, Glk can recognize when the mouse (or other pointer) is used to select a spot in a window. You can request mouse input only in text grid windows and graphics windows.

```
   void glk_request_mouse_event(winid_t win);
   void glk_cancel_mouse_event(winid_t win);
```

A window can have mouse input and character/line input pending at the same time.

If the player clicks in a window which has a mouse input event pending, glk_select() will return an event whose type is evtype_MouseInput. Again, once this happens, the request is complete, and you must request another if you want further mouse input.

In the event structure, win tells what window the event came from.

In a text grid window, the val1 and val2 fields are the x and y coordinates of the character that was clicked on. *[So val1 is the column, and val2 is the row.]* The top leftmost character is considered to be (0,0).

In a graphics window, they are the x and y coordinates of the pixel that was clicked on. Again, the top left corner of the window is (0,0).

You can test whether mouse input is supported with the gestalt_MouseInput selector.

```
   res = glk_gestalt(gestalt_MouseInput, windowtype);
```

This will return TRUE (1) if windows of the given type support mouse input. If this returns FALSE (0), it is still legal to call glk_request_mouse_event(), but it will have no effect, and you will never get mouse events.

[Most mouse-based idioms define standard functions for mouse hits in text windows — typically selecting or copying text. It is up to the library to separate this from Glk mouse input. The library may choose to select text when it is clicked normally, and cause Glk mouse events when text is control-clicked. Or the other way around. Or it may be the difference between clicking and double-clicking. Or the library may reserve a particular mouse button, on a multi-button mouse. It may even specify a keyboard key to be the "mouse button", referring to wherever the mouse cursor is when the key is hit. Or some even more esoteric positioning system. You need only know that the user can do it, or not.]

[However, since different platforms will handle this issue differently, you should be careful how you instruct the player in your program. Do not tell the player to "double-click", "right-click", or "control-click" in a window. The preferred term is "to touch the window", or a spot in the window.] [Goofy, but preferred.]

4.4. Timer Events

You can request that an event be sent at fixed intervals, regardless of what the player does. Unlike input events, timer events can be tested for with glk_select_poll() as well as glk_select().

```
   void glk_request_timer_events(glui32 millisecs);
```

It is possible that the library does not support timer events. You can check this with the gestalt_Timer selector.

```
   res = glk_gestalt(gestalt_Timer, 0);
```

This returns TRUE (1) if timer events are supported, and FALSE (0) if they are not.

Initially, there is no timer and you get no timer events. If you call glk_request_timer_events(N), with N not 0, you will get timer events about every N milliseconds thereafter. (Assuming that they are supported — if not, glk_request_timer_events() has no effect.) Unlike keyboard and mouse events, timer events will continue until you shut them off. You do not have to re-request them every time you get one. Call glk_request_timer_events(0) to stop getting timer events.

The rule is that when you call glk_select() or glk_select_poll(), if it has been more than N milliseconds since the last timer event, and (for glk_select()) if there is no player input, you will receive an event whose type is evtype_Timer. (win, val1, and val2 will all be 0.)

Timer events do not stack up. If you spend 10N milliseconds doing computation, and then call glk_select(), you will not get ten timer events in a row. The library will simply note that it has been more than N milliseconds, and return a timer event right away. If you call glk_select() again immediately, it will be N milliseconds before the next timer event.

This means that the timing of timer events is approximate, and the library will err on the side of being late. If there is a conflict between player input events and timer events, the player input takes precedence. *[This prevents the user from being locked out by overly enthusiastic timer events. Unfortunately, it also means that your timer can be locked out on slower machines, if the player pounds too enthusiastically on the keyboard. Sorry.]*

[I don't have to tell you that a millisecond is one thousandth of a second, do I?]

4.5. Window Arrangement Events

Some platforms allow the player to resize the Glk window during play. This will naturally change the sizes of your windows. If this occurs, then immediately *after* all the rearrangement, glk_select() will return an event whose type is evtype_Arrange. You can use this notification to redisplay the contents of a graphics or text grid window whose size has changed. *[The display of a text buffer window is entirely up to the library, so you don't need to worry about those.]*

In the event structure, win will be NULL if all windows are affected. If only some windows are affected, win will refer to a window which contains all the affected windows. *[You can always play it safe, ignore win, and redraw every graphics and text grid window.]* val1 and val2 will be 0.

An arrangement event is guaranteed to occur whenever the player causes any window to change size, as measured by its own metric. *[Size changes caused by you — for example, if you open, close, or resize a window — do not trigger arrangement events. You must be aware of the effects of your window management, and redraw the windows that you affect.]*

[It is possible that several different player actions can cause windows to change size. For example, if the player changes the screen resolution, an arrangement event might be triggered. This might also happen if the player changes his display font to a different size; the windows would then be different "sizes" in the metric of rows and columns, which is the important metric and the only one you have access to.]

Arrangement events, like timer events, can be returned by glk_select_poll(). But this will not occur on all platforms. You must be ready to receive an arrangement event when you call glk_select_poll(), but it is possible that it will not arrive until the next time you call glk_select(). *[This is because on some platforms, window resizing is handled as part of player input; on others, it can be triggered by an external process such as a window manager.]*

4.6. Window Redrawing Events

On platforms that support graphics, it is possible that the contents of a graphics window will be lost, and have to be redrawn from scratch. If this occurs, then glk_select() will return an event whose type is evtype_Redraw.

In the event structure, win will be NULL if all windows are affected. If only some windows are affected, win will refer to a window which contains all the affected windows. *[You can always play it safe, ignore win, and redraw every graphics window.]* val1 and val2 will be 0.

Affected windows are already cleared to their background color when you receive the redraw event.

Redraw events can be returned by glk_select_poll(). But, like arrangement events, this is platform-dependent. See section 4.5, "Window Arrangement Events".

For more about redraw events and how they affect graphics windows, see section 3.5.5, "Graphics Windows".

4.7. Sound Notification Events

On platforms that support sound, you can request to receive an evtype_SoundNotify event when a sound finishes playing. You can also request to receive an evtype_VolumeNotify event when a gradual volume change completes. See section 8.3, "Playing Sounds".

4.8. Hyperlink Events

On platforms that support hyperlinks, you can request to receive an evtype_Hyperlink event when the player selects a link. See section 9.2, "Accepting Hyperlink Events".

4.9. Other Events

There are currently no other event types defined by Glk. (The "evtype_None" constant is a placeholder, and is never returned by glk_select().)

It is possible that new event types will be defined in the future. *[For example, if video or animation capabilities are added to Glk, there would probably be some sort of completion event for them.]*

[This is also why you must put calls to glk_select() in loops. If you tried to read a character by simply writing

```
glk_request_char_event(win);
glk_select(&ev);
```

you might not get a CharInput event back. You could get some not-yet-defined event which happened to occur before the player hit a key. Or, for that matter, a window arrangement event.]

[It is not generally necessary to put a call to glk_select_poll() in a loop. You usually call glk_select_poll() to update the display or test if an event is available, not to wait for a particular event. However, if you are using sound notification events, and several sounds are playing, it might be important to make sure you knew about all sounds completed at any particular time. You would do this with

```
glk_select_poll(&ev);
while (ev.type != evtype_None) {
    /* handle event */
    glk_select_poll(&ev);
}
```

Once glk_select_poll() returns evtype_None, you should not *call it again immediately. Do some work first. If you want to wait for an event, use glk_select(), not a loop of glk_select_poll().]*

5. Streams

All character output in Glk is done through streams. Every window has an output stream associated with it. You can also write to files on disk; every open file is represented by an output stream as well.

There are also input streams; these are used for reading from files on disk. It is possible for a stream to be both an input and an output stream. *[Player input is done through line and character input events, not streams. This is a small inelegance in theory. In practice, player input is slow and things can interrupt it, whereas file input is immediate. If a network extension to Glk were proposed, it would probably use events and not* *streams, since network communication is not immediate.]*

It is also possible to create a stream that reads or writes to a buffer in memory.

Finally, there may be platform-specific types of streams, which are created before your program starts running. *[For example, a program running under Unix may have access to standard input as a stream, even though there is no Glk call to explicitly open standard input. On the Mac, data in a Mac resource may be available through a resource-reading stream.]* You do not need to worry about the origin of such streams; just read or write them as usual. For information about how platform-specific streams come to be, see section 11.1, "Startup Options".

A stream is opened with a particular file mode:

- filemode_Write: An output stream.
- filemode_Read: An input stream.
- filemode_ReadWrite: Both an input and an output stream.
- filemode_WriteAppend: An output stream, but the data will added to the end of whatever already existed in the destination, instead of replacing it.

[In the stdio library, using fopen(), filemode_Write would be mode "w"; filemode_Read would be mode "r"; filemode_ReadWrite would be mode "r+". Confusingly, filemode_WriteAppend cannot be mode "a", because the stdio spec says that when you open a file with mode "a", then fseek() doesn't work. So we have to use mode "r+" for appending. Then we run into the other *stdio problem, which is that "r+" never creates a new file. So filemode_WriteAppend has to* first *open the file with "a", close it, reopen with "r+", and then fseek() to the end of the file. For filemode_ReadWrite, the process is the same, except without the fseek() — we begin at the beginning of the file.]*

[We must also obey an obscure geas of ANSI C "r+" files: you can't switch from reading to writing without doing an fseek() in between. Switching from writing to reading has the same restriction, except that an fflush() also works.]

For information on opening streams, see the discussion of each specific type of stream in section 5.6, "The Types of Streams". Remember that it is always possible that opening a stream will fail, in which case the creation function will return NULL.

Each stream remembers two character counts, the number of characters printed to and read from that stream. The write-count is exactly one per glk_put_char() call; it is figured before any platform-dependent character cookery. *[For example, if a newline character is converted to linefeed-plus-carriage-return, the stream's count still only goes up by one; similarly if an accented character is displayed as two characters.]* The read-count is exactly one per glk_get_char_stream() call, as long as the call returns an actual character (as opposed to an end-of-file token.)

Glk has a notion of the "current (output) stream". If you print text without specifying a stream, it goes to the current output stream. The current output stream may be NULL, meaning that there isn't one. It is illegal to print text to stream NULL, or to print to the current stream when there isn't one.

If the stream which is the current stream is closed, the current stream becomes NULL.

```
void glk_stream_set_current(strid_t str);
```

This sets the current stream to str, which must be an output stream. You may set the current stream to NULL, which means the current stream is not set to anything.

```
strid_t glk_stream_get_current(void);
```

Returns the current stream, or NULL if there is none.

5.1. How To Print

```
void glk_put_char(unsigned char ch);
```

This prints one character to the current stream. As with all basic functions, the character is assumed to be in the Latin-1 character encoding. See section 2, "Character Encoding".

```
void glk_put_string(char *s);
```

This prints a null-terminated string to the current stream. It is exactly equivalent to

```
for (ptr = s; *ptr; ptr++)
    glk_put_char(*ptr);
```

However, it may be more efficient.

```
void glk_put_buffer(char *buf, glui32 len);
```

This prints a block of characters to the current stream. It is exactly equivalent to

```
for (i = 0; i < len; i++)
    glk_put_char(buf[i]);
```

However, it may be more efficient.

```
void glk_put_char_stream(strid_t str, unsigned char ch);
void glk_put_string_stream(strid_t str, char *s);
void glk_put_buffer_stream(strid_t str, char *buf, glui32 len);
```

These are the same functions, except that you specify a stream to print to, instead of using the current stream. Again, it is illegal for str to be NULL, or the reference of an input-only stream.

```
void glk_put_char_uni(glui32 ch);
```

This prints one character to the current stream. The character is assumed to be a Unicode code point. See section 2, "Character Encoding".

```
void glk_put_string_uni(glui32 *s);
```

This prints a string of Unicode characters to the current stream. It is equivalent to a series of glk_put_char_uni() calls. A string ends on a glui32 whose value is 0.

```
void glk_put_buffer_uni(glui32 *buf, glui32 len);
```

This prints a block of Unicode characters to the current stream. It is equivalent to a series of glk_put_char_uni() calls.

```
void glk_put_char_stream_uni(strid_t str, glui32 ch);
void glk_put_string_stream_uni(strid_t str, glui32 *s);
void glk_put_buffer_stream_uni(strid_t str, glui32 *buf, glui32 len);
```

5.2. How To Read

```
glsi32 glk_get_char_stream(strid_t str);
```

This reads one character from the given stream. (There is no notion of a "current input stream.") It is illegal for str to be NULL, or an output-only stream.

The result will be between 0 and 255. As with all basic text functions, Glk assumes the Latin-1 encoding. See section 2, "Character Encoding". If the end of the stream has been reached, the result will be -1. *[Note that high-bit characters (128..255) are* not *returned as negative numbers.]*

If the stream contains Unicode data — for example, if it was created with glk_stream_open_file_uni() or glk_stream_open_memory_uni() — then characters beyond 255 will be returned as 0x3F ("?").

```
glui32 glk_get_buffer_stream(strid_t str, char *buf, glui32 len);
```

This reads len characters from the given stream, unless the end of stream is reached first. No terminal null is placed in the buffer. It returns the number of characters actually read.

```
glui32 glk_get_line_stream(strid_t str, char *buf, glui32 len);
```

This reads characters from the given stream, until either len-1 characters have been read or a newline has been read. It then puts a terminal null ('\0') character on the end. It returns the number of characters actually read, including the newline (if there is one) but not including the terminal null.

It is usually more efficient to read several characters at once with glk_get_buffer_stream() or glk_get_line_stream(), as opposed to calling glk_get_char_stream() several times.

```
glsi32 glk_get_char_stream_uni(strid_t str);
```

Reads one character from the given stream. If the end of the stream has been reached, the result will be -1.

```
glui32 glk_get_buffer_stream_uni(strid_t str, glui32 *buf, glui32 len);
```

This reads len Unicode characters from the given stream, unless the end of the stream is reached first. No terminal null is placed in the buffer. It returns the number of Unicode characters actually read.

```
glui32 glk_get_line_stream_uni(strid_t str, glui32 *buf, glui32 len);
```

This reads Unicode characters from the given stream, until either len-1 Unicode characters have been read or a newline has been read. It then puts a terminal null (a zero value) on the end. It returns the number of Unicode characters actually read, including the newline (if there is one) but not including the terminal null.

5.3. Closing Streams

```
void glk_stream_close(strid_t str, stream_result_t *result);


typedef struct stream_result_struct {
    glui32 readcount;
    glui32 writecount;
} stream_result_t;
```

This closes the stream str. The result argument points to a structure which is filled in with the final character counts of the stream. If you do not care about these, you may pass NULL as the result argument.

If str is the current output stream, the current output stream is set to NULL.

You cannot close window streams; use glk_window_close() instead. See section 3.2, "Window Opening, Closing, and Constraints".

5.4. Stream Positions

You can set the position of the read/write mark in a stream. *[Which makes one wonder why they're called "streams" in the first place. Oh well.]*

```
    glui32 glk_stream_get_position(strid_t str);
```

This returns the position of the mark. For memory streams and binary file streams, this is exactly the number of characters read or written from the beginning of the stream (unless you have moved the mark with glk_stream_set_position().) For text file streams, matters are more ambiguous, since (for example) writing one byte to a text file may store more than one character in the platform's native encoding. You can only be sure that the position increases as you read or write to the file.

Additional complication: for Latin-1 memory and file streams, a character is a byte. For Unicode memory and file streams (those created by glk_stream_open_file_uni() and glk_stream_open_memory_uni()), a character is a 32-bit word. So in a binary Unicode file, positions are multiples of four bytes.

[If this bothers you, don't use binary Unicode files. I don't think they're good for much anyhow.]

```
    void glk_stream_set_position(strid_t str, glsi32 pos, glui32 seekmode);
```

This sets the position of the mark. The position is controlled by pos, and the meaning of pos is controlled by seekmode:

- seekmode_Start: pos characters after the beginning of the file.
- seekmode_Current: pos characters after the current position (moving backwards if pos is negative.)
- seekmode_End: pos characters after the end of the file. (pos should always be zero or negative, so that this will move backwards to a position within the file.)

It is illegal to specify a position before the beginning or after the end of the file.

In binary files, the mark position is exact — it corresponds with the number of characters you have read or written. In text files, this mapping can vary, because of linefeed conversions or other character-set approximations. (See section 5, "Streams".) glk_stream_set_position() and glk_stream_get_position() measure positions in the platform's native encoding — after character cookery. Therefore, in a text stream, it is safest to use glk_stream_set_position() only to move to the beginning or end of a file, or to a position determined by glk_stream_get_position().

Again, in Latin-1 streams, characters are bytes. In Unicode streams, characters are 32-bit words, or four bytes each.

A window stream doesn’t have a movable mark, so calling glk_stream_set_position() has no effect. glk_stream_get_position() on a window stream will always return zero. *[It might make more sense to return the number of characters written to the window, but existing libraries do not support this and it’s not really worth adding the feature.]*

5.5. Styles

You can send style-changing commands to an output stream. After a style change, new text which is printed to that stream will be given the new style, whatever that means for the stream in question. For a window stream, the text will appear in that style. For a memory stream, style changes have no effect. For a file stream, if the machine supports styled text files, the styles may be written to the file; more likely the style changes will have no effect.

Styles are exclusive. A character is shown with exactly one style, not a subset of the possible styles.

[Note that every stream and window has its own idea of the "current style." Sending a style command to one window or stream does not affect any others.] [Except for a window’s echo stream; see section 3.6, "Echo Streams".]

The styles are intended to distinguish meaning and use, not formatting. There is *no* standard definition of what each style will look like. That is left up to the Glk library, which will choose an appearance appropriate for the platform’s interface and the player’s preferences.

There are currently eleven styles defined. More may be defined in the future.

- style_Normal: The style of normal or body text. A new window or stream always starts with style_Normal as the current style.
- style_Emphasized: Text which is emphasized.
- style_Preformatted: Text which has a particular arrangement of characters. *[This style, unlike the others,* does *have a standard appearance; it will always be a fixed-width font. This is a concession to practicality. Games often want to display maps or diagrams using character graphics, and this is the style for that.]*
- style_Header: Text which introduces a large section. This is suitable for the title of an entire game, or a major division such as a chapter.
- style_Subheader: Text which introduces a smaller section within a large section. *[In a Colossal-Cave-style game, this is suitable for the name of a room (when the player looks around.)]*
- style_Alert: Text which warns of a dangerous condition, or one which the player should pay attention to.
- style_Note: Text which notifies of an interesting condition. *[This is suitable for noting that the player’s score has changed.]*
- style_BlockQuote: Text which forms a quotation or otherwise abstracted text.
- style_Input: Text which the player has entered. You should generally not use this style at all; the library uses it for text which is typed during a line-input request. One case when it *is* appropriate for you to use style_Input is when you are simulating player input by reading commands from a text file.
- style_User1: This style has no particular semantic meaning. You may define a meaning relevant to your own work, and use it as you see fit.
- style_User2: Another style available for your use.

Styles may be distinguished on screen by font, size, color, indentation, justification, and other attributes. Note that some attributes (notably justification and indentation) apply to entire paragraphs. If possible and relevant, you should apply a style to an entire paragraph — call glk_set_style() immediately after printing the newline at the beginning of the text, and do the same at the end.

[For example, style_Header may well be centered text. If you print "Welcome to Victim (a short interactive mystery)", and only the word "Victim" is in the style_Header, the center-justification attribute will be lost. Similarly, a block quote is usually indented on both sides, but indentation is only meaningful when applied to an entire line or paragraph, so block quotes should take up an entire paragraph. Contrariwise, style_Emphasized need not be used on an entire paragraph. It is often used for single emphasized words in normal text, so you can expect that it will appear properly that way; it will be displayed in italics or underlining, not center-justified or indented.]

[Yes, this is all a matter of mutual agreement between game authors and game players. It's not fixed by this specification. That's natural language for you.]

```
   void glk_set_style(glui32 val);
```

This changes the style of the current output stream. val should be one of the values listed above. However, any value is actually legal; if the interpreter does not recognize the style value, it will treat it as style_Normal. *[This policy allows for the future definition of styles without breaking old Glk libraries.]*

```
   void glk_set_style_stream(strid_t str, glui32 val);
```

This changes the style of the stream str.

5.5.1. Suggesting the Appearance of Styles

There are no guarantees of how styles will look, but you can make suggestions.

```
   void glk_stylehint_set(glui32 wintype, glui32 styl, glui32 hint, glsi32
      val);
   void glk_stylehint_clear(glui32 wintype, glui32 styl, glui32 hint);
```

These functions set and clear hints about the appearance of one style for a particular type of window. You can also set wintype to wintype_AllTypes, which sets (or clears) a hint for all types of window. *[There is no equivalent constant to set a hint for all styles of a single window type.]*

Initially, no hints are set for any window type or style. Note that having no hint set is not the same as setting a hint with value 0.

These functions do *not* affect *existing* windows. They affect the windows which you create subsequently. If you want to set hints for all your game windows, call glk_stylehint_set() before you start creating windows. If you want different hints for different windows, change the hints before creating each window.

[This policy makes life easier for the interpreter. It knows everything about a particular window's appearance when the window is created, and it doesn't have to change it while the window exists.]

Hints are hints. The interpreter may ignore them, or give the player a choice about whether to accept them. Also, it is never necessary to set hints. You don't have to suggest that style_Preformatted be fixed-width, or style_Emphasized be boldface or italic; they will have appropriate defaults. Hints are for situations when you want to *change* the appearance of a style from what it would ordinarily be. The most common case when this is appropriate is for the styles style_User1 and style_User2.

There are currently nine style hints defined. More may be defined in the future.

- stylehint_Indentation: How much to indent lines of text in the given style. May be a negative number, to shift the text out (left) instead of in (right). The exact metric isn't precisely specified; you can assume that +1 is the smallest indentation possible which is clearly visible to the player.
- stylehint_ParaIndentation: How much to indent the first line of each paragraph. This is in addition to the indentation specified by stylehint_Indentation. This too may be negative, and is measured in the same units as stylehint_Indentation.
- stylehint_Justification: The value of this hint must be one of the constants stylehint_just_LeftFlush, stylehint_just_LeftRight (full justification), stylehint_just_Centered, or stylehint_just_RightFlush.
- stylehint_Size: How much to increase or decrease the font size. This is relative; 0 means the interpreter's default font size will be used, positive numbers increase it, and negative numbers decrease it. Again, +1 is the smallest size increase which is easily visible. *[The amount of this increase may not be constant. +1 might increase an 8-point font to 9-point, but a 16-point font to 18-point.]*
- stylehint_Weight: The value of this hint must be 1 for heavy-weight fonts (boldface), 0 for normal weight, and -1 for light-weight fonts.
- stylehint_Oblique: The value of this hint must be 1 for oblique fonts (italic), or 0 for normal angle.
- stylehint_Proportional: The value of this hint must be 1 for proportional-width fonts, or 0 for fixed-width.
- stylehint_TextColor: The foreground color of the text. This is encoded in the 32-bit hint value: the top 8 bits must be zero, the next 8 bits are the red value, the next 8 bits are the green value, and the bottom 8 bits are the blue value. Color values range from 0 to 255. *[So 0x00000000 is black, 0x00FFFFFF is white, and 0x00FF0000 is bright red.]*
- stylehint_BackColor: The background color behind the text. This is encoded the same way as stylehint_TextColor.
- stylehint_ReverseColor: The value of this hint must be 0 for normal printing (TextColor on BackColor), or 1 for reverse printing (BackColor on TextColor). *[Some libraries may support this hint but not the TextColor and BackColor hints. Other libraries may take the opposite tack; others may support both, or neither.]*

Again, when passing a style hint to a Glk function, any value is actually legal. If the interpreter does not recognize the stylehint value, it will ignore it. *[This policy allows for the future definition of style hints without breaking old Glk libraries.]*

5.5.2. Testing the Appearance of Styles

You can suggest the appearance of a window's style before the window is created; after the window is created, you can test the style's actual appearance. These functions do not test the style hints; they test the attribute of the style as it appears to the player.

Note that although you cannot change the appearance of a window's styles after the window is created, the library can. A platform may support dynamic preferences, which allow the player to change text formatting while your program is running. *[Changes that affect window size (such as font size changes) will be signalled by an evtype_Arrange event. However, more subtle changes (such as text color differences) are not signalled. If you test the appearance of styles at the beginning of your program, you must keep in mind the possibility that the player will change them later.]*

```
glui32 glk_style_distinguish(winid_t win, glui32 styl1, glui32 styl2);
```

This returns TRUE (1) if the two styles are visually distinguishable in the given window. If they are not, it returns FALSE (0). The exact meaning of this is left to the library to determine.

```
glui32 glk_style_measure(winid_t win, glui32 styl, glui32 hint, glui32
      *result);
```

This tries to test an attribute of one style in the given window. The library may not be able to determine the attribute; if not, this returns FALSE (0). If it can, it returns TRUE (1) and stores the value in the location pointed at by result. *[As usual, it is legal for result to be NULL, although fairly pointless.]*

The meaning of the value depends on the hint which was tested:

- stylehint_Indentation, stylehint_ParaIndentation: The indentation and paragraph indentation. These are in a metric which is platform-dependent. *[Most likely either characters or pixels.]*
- stylehint_Justification: One of the constants stylehint_just_LeftFlush, stylehint_just_LeftRight, stylehint_just_Centered, or stylehint_just_RightFlush.
- stylehint_Size: The font size. Again, this is in a platform-dependent metric. *[Pixels, points, or simply 1 if the library does not support varying font sizes.]*
- stylehint_Weight: 1 for heavy-weight fonts (boldface), 0 for normal weight, and -1 for light-weight fonts.
- stylehint_Oblique: 1 for oblique fonts (italic), or 0 for normal angle.
- stylehint_Proportional: 1 for proportional-width fonts, or 0 for fixed-width.
- stylehint_TextColor, stylehint_BackColor: These are values from 0x00000000 to 0x00FFFFFF, encoded as described in section 5.5.1, "Suggesting the Appearance of Styles".
- stylehint_ReverseColor: 0 for normal printing, 1 if the foreground and background colors are reversed.

Signed values, such as the stylehint_Weight value, are cast to glui32. They may be cast to glsi32 to be dealt with in a more natural context.

5.6. The Types of Streams

5.6.1. Window Streams

Every window has an output stream associated with it. This is created automatically, with filemode_Write, when you open the window. You get it with glk_window_get_stream(). Window streams always have rock value 0.

A window stream cannot be closed with glk_stream_close(). It is closed automatically when you close its window with glk_window_close().

Only printable characters (including newline) may be printed to a window stream. See section 2, "Character Encoding".

5.6.2. Memory Streams

You can open a stream which reads from or writes into a space in memory.

```
strid_t glk_stream_open_memory(char *buf, glui32 buflen, glui32 fmode,
   glui32 rock);
```

fmode must be filemode_Read, filemode_Write, or filemode_ReadWrite.

buf points to the buffer where output will be read from or written to. buflen is the length of the buffer.

When outputting, if more than buflen characters are written to the stream, all of them beyond the buffer length will be thrown away, so as not to overwrite the buffer. (The character count of the stream will still be maintained correctly. That is, it will count the number of characters written into the stream, not the number that fit in the buffer.)

If buf is NULL, or for that matter if buflen is zero, then *everything* written to the stream is thrown away. This may be useful if you are interested in the character count.

When inputting, if more than buflen characters are read from the stream, the stream will start returning -1 (signalling end-of-file.) If buf is NULL, the stream will always return end-of-file.

The data is written to the buffer exactly as it was passed to the printing functions (glk_put_char(), etc); input functions will read the data exactly as it exists in memory. No platform-dependent cookery will be done on it. *[You can write a disk file in text mode, but a memory stream is effectively always in binary mode.]*

Unicode values (characters greater than 255) cannot be written to the buffer. If you try, they will be stored as 0x3F ("?") characters.

Whether reading or writing, the contents of the buffer are undefined until the stream is closed. The library may store the data there as it is written, or deposit it all in a lump when the stream is closed. It is illegal to change the contents of the buffer while the stream is open.

```
strid_t glk_stream_open_memory_uni(glui32 *buf, glui32 buflen, glui32
   fmode, glui32 rock);
```

This works just like glk_stream_open_memory(), except that the buffer is an array of 32-bit words, instead of bytes. This allows you to write and read any Unicode character. The buflen is the number of words, not the number of bytes.

[If the buffer contains the value 0xFFFFFFFF, and is opened for reading, the reader cannot distinguish that value from -1 (end-of-file). Fortunately 0xFFFFFFFF is not a valid Unicode character.]

5.6.3. File Streams

You can open a stream which reads from or writes to a disk file.

```
strid_t glk_stream_open_file(frefid_t fileref, glui32 fmode, glui32 rock);
```

fileref indicates the file which will be opened. fmode can be any of filemode_Read, filemode_Write, filemode_WriteAppend, or filemode_ReadWrite. If fmode is filemode_Read, the file must already exist; for the other modes, an empty file is created if none exists. If fmode is filemode_Write, and the file already exists, it is truncated down to zero length (an empty file); the other modes do not truncate. If fmode is filemode_WriteAppend, the file mark is set to the end of the file.

[Note, again, that this doesn't match stdio's fopen() call very well. See section 5, "Streams"]

If the filemode requires the file to exist, but the file does not exist, glk_stream_open_file() returns NULL.

The file may be read or written in text or binary mode; this is determined by the fileref argument. Similarly, platform-dependent attributes such as file type are determined by fileref. See section 6, "File References".

When writing in binary mode, Unicode values (characters greater than 255) cannot be written to the file. If you try, they will be stored as 0x3F ("?") characters. In text mode, Unicode values may be stored exactly, approximated, or abbreviated, depending on what the platform’s text files support.

```
   strid_t glk_stream_open_file_uni(frefid_t fileref, glui32 fmode, glui32
      rock);
```

This works just like glk_stream_open_file(), except that in binary mode, characters are written and read as four-byte (big-endian) values. This allows you to write and read any Unicode character.

In text mode, the file is written and read in a platform-dependent way, which may or may not handle all Unicode characters. A text-mode file created with glk_stream_open_file_uni() may have the same format as a text-mode file created with glk_stream_open_file(); or it may use a more Unicode-friendly format.

5.6.4. Resource Streams

You can open a stream which reads from (but not writes to) a resource file.

[Typically this is embedded in a Blorb file, as Blorb is the official resource-storage format of Glk. A Blorb file can contain images and sounds, but it can also contain raw data files, which are accessed by the following functions. A data file is identified by number, not by a filename. The Blorb usage field will be ’Data’. The chunk type will be ’TEXT’ for text resources, ’BINA’ for binary resources.]

[If the running program is not associated with a Blorb file, the library may look for data files as actual files instead. These would be named "DATA1", "DATA2", etc, with a suffix distinguishing text and binary files. See "Other Resource Arrangements" in the Blorb spec: <http://eblong.com/zarf/blorb/>]

```
   strid_t glk_stream_open_resource(glui32 filenum, glui32 rock);
   strid_t glk_stream_open_resource_uni(glui32 filenum, glui32 rock);
```

Open the given data resource for reading (only), as a normal or Unicode stream. *[Note that there is no notion of file usage -- the resource does not have to be specified as "saved game" or whatever.]*

If no resource chunk of the given number exists, the open function returns NULL.

As with file streams, a binary resource stream reads the resource as bytes (for a normal stream) or as four-byte (big-endian) words (for a Unicode stream). A text resource stream reads characters encoded as Latin-1 (for normal) or UTF-8 (for Unicode). *[Thus, the difference between text and binary resources is only important when opened as a Unicode stream.]*

When reading from a resource stream, newlines are not remapped, even if they normally would be when reading from a text file on the host OS. If you read a line (glk_get_line_stream or glk_get_line_stream_uni), a Unix newline (0x0A) terminates the line.

```
   res = glk_gestalt(gestalt_ResourceStream, 0);
```

This returns 1 if the glk_stream_open_resource() and glk_stream_open_resource_uni() functions are available. If it returns 0, you should not call them.

5.7. Other Stream Functions

```
   strid_t glk_stream_iterate(strid_t str, glui32 *rockptr);
```

This iterates through all the existing streams. See section 1.6.2, "Iterating Through Opaque Objects".

```
   glui32 glk_stream_get_rock(strid_t str);
```

This retrieves the stream's rock value. See section 1.6.1, "Rocks". Window streams always have rock 0; all other streams return whatever rock you created them with.

6. File References

You deal with disk files using file references. Each fileref is an opaque C structure pointer; see section 1.6, "Opaque Objects".

A file reference contains platform-specific information about the name and location of the file, and possibly its type, if the platform has a notion of file type. It also includes a flag indication whether the file is a text file or binary file. *[Note that this is different from the standard C I/O library, in which you specify text or binary mode when the file is opened.]*

A fileref does not have to refer to a file which actually exists. You can create a fileref for a nonexistent file, and then open it in write mode to create a new file.

You always provide a usage argument when you create a fileref. The usage indicates the file type and the mode (text or binary.) It must be the logical-or of a file-type constant and a mode constant. These values are used when you create a new file, and also to filter file lists when the player is selecting a file to load. The constants are as follows:

- fileusage_SavedGame: A file which stores game state.
- fileusage_Transcript: A file which contains a stream of text from the game (often an echo stream from a window.)
- fileusage_InputRecord: A file which records player input.
- fileusage_Data: Any other kind of file (preferences, statistics, arbitrary data.)

- fileusage_BinaryMode: The file contents will be stored exactly as they are written, and read back in the same way. The resulting file may not be viewable on platform-native text file viewers.
- fileusage_TextMode: The file contents will be transformed to a platform-native text file as they are written out. Newlines may be converted to linefeeds or linefeed-plus-carriage-return combinations; Latin-1 characters may be converted to native character codes. When reading a file in text mode, native line breaks will be converted back to newline (0x0A) characters, and native character codes may be converted to Latin-1 or UTF-8. *[Line breaks will always be converted; other conversions are more questionable. If you write out a file in text mode, and then read it back in text mode, high-bit characters (128 to 255) may be transformed or lost.]*

In general, you should use text mode if the player expects to read the file with a platform-native text editor; you should use binary mode if the file is to be read back by your program, or if the data must be stored exactly. Text mode is appropriate for fileusage_Transcript; binary mode is appropriate for fileusage_SavedGame and probably for fileusage_InputRecord. fileusage_Data files may be text or binary, depending on what you use them for.

6.1. The Types of File References

There are four different functions for creating a fileref, depending on how you wish to specify it. Remember that it is always possible that a fileref creation will fail and return NULL.

```
   frefid_t glk_fileref_create_temp(glui32 usage, glui32 rock);
```

This creates a reference to a temporary file. It is always a new file (one which does not yet exist). The file (once created) will be somewhere out of the player's way. *[This is why no name is specified; the player will never need to know it.]*

A temporary file should not be used for long-term storage. It may be deleted automatically when the program exits, or at some later time, say when the machine is turned off or rebooted. You do not have to worry about deleting it yourself.

```
   frefid_t glk_fileref_create_by_prompt(glui32 usage, glui32 fmode, glui32
      rock);
```

This creates a reference to a file by asking the player to locate it. The library may simply prompt the player to type a name, or may use a platform-native file navigation tool. (The prompt, if any, is inferred from the usage argument.)

fmode must be one of these values:

- filemode_Read: The file must already exist; the player will be asked to select from existing files which match the usage.
- filemode_Write: The file should not exist; if the player selects an existing file, he will be warned that it will be replaced.
- filemode_ReadWrite: The file may or may not exist; if it already exists, the player will be warned that it will be modified.
- filemode_WriteAppend: Same behavior as filemode_ReadWrite.

The fmode argument should generally match the fmode which will be used to open the file.

[It is likely that the prompt or file tool will have a "cancel" option. If the player chooses this, glk_fileref_create_by_prompt() will return NULL. This is a major reason why you should make sure the return value is valid before you use it.]

The recommended file suffixes for files are ".glkdata" for fileusage_Data, ".glksave" for fileusage_SavedGame, ".txt" for fileusage_Transcript and fileusage_InputRecord.

```
   frefid_t glk_fileref_create_by_name(glui32 usage, char *name, glui32 rock);
```

This creates a reference to a file with a specific name. The file will be in a fixed location relevant to your program, and visible to the player. *[This usually means "in the same directory as your program."]*

Earlier versions of the Glk spec specified that the library may have to extend, truncate, or change your name argument in order to produce a legal native filename. This remains true. However, since Glk was originally proposed, the world has largely reached concensus about what a filename looks like. Therefore, it is worth including some recommended library behavior here. Libraries that share this behavior will more easily be able to exchange files, which may be valuable both to authors (distributing data files for games) and for players (moving data between different computers or different applications).

The library should take the given filename argument, and delete any characters illegal for a filename. This will include all of the following characters (and more, if the OS requires it): slash, backslash, angle brackets (less-than and greater-than), colon, double-quote, pipe (vertical bar), question-mark, asterisk. The library should also truncate the argument at the first period (delete the first period and any following characters). If the result is the empty string, change it to the string "null".

It should then append an appropriate suffix, depending on the usage: ".glkdata" for fileusage_Data, ".glksave" for fileusage_SavedGame, ".txt" for fileusage_Transcript and fileusage_InputRecord.

The above behavior is *not* a requirement of the Glk spec. Older implementations can continue doing what they do. Some programs (e.g. web-based interpreters) may not have access to a traditional filesystem at all, and to them these recommendations will be meaningless.

On the other side of the coin, the game file should not press these limitations. Best practice is for the game to pass a filename containing only letters and digits, beginning with a letters, and not mixing upper and lower case. Avoid overly-long filenames.

[The earlier Glk spec gave more stringent recommendations: "No more than 8 characters, consisting entirely of upper-case letters and numbers, starting with a letter". The DOS era is safely contained, if not over, so this has been relaxed. The I7 manual recommends "23 characters or fewer".]

[To address other complications:]

[Some filesystems are case-insensitive. If you create two filerefs with the names "File" and "FILE", they may wind up pointing to the same file, or they may not. Avoid doing this.]

[Some programs will look for all files in the same directory as the program itself (or, for interpreted games, in the same directory as the game file). Others may keep files in a data-specific directory appropriate for the user (e.g., ~/Library on MacOS).]

[If a game interpreter uses a data-specific directory, there is a question of whether to use a common location, or divide it into game-specific subdirectories. (Or to put it another way: should the namespace of named files be per-game or app-wide?) Since data files may be exchanged between games, they should be given an app-wide namespace. In contrast, saved games should be per-game, as they can never be exchanged. Transcripts and input records can go either way.]

[When updating an older library to follow these recommendations, consider backwards compatibility for games already installed. When opening an existing file (that is, not in a write-only mode) it may be worth looking under the older name (suffix) if the newer one does not already exist.]

[Game-save files are already stored with a variety of file suffixes, since that usage goes back to the oldest IF interpreters, long predating Glk. It is reasonable to treat them in some special way, while hewing closer to these recommendations for data files.]

```
frefid_t glk_fileref_create_from_fileref(glui32 usage, frefid_t fref,
   glui32 rock);
```

This copies an existing file reference, but changes the usage. (The original fileref is not modified.)

The use of this function can be tricky. If you change the type of the fileref (fileusage_Data, fileusage_SavedGame, etc), the new reference may or may not point to the same actual disk file. *[Most platforms use suffixes to indicate file type, so it typically will not. See the earlier comments about recommended file suffixes.]* If you do this, and open both file references for writing, the results are unpredictable. It is safest to change the type of a fileref only if it refers to a nonexistent file.

If you change the mode of a fileref (fileusage_TextMode, fileusage_BinaryMode), but leave the rest of the type unchanged, the new fileref will definitely point to the same disk file as the old one.

Obviously, if you write to a file in text mode and then read from it in binary mode, the results are platform-dependent.

6.2. Other File Reference Functions

```
void glk_fileref_destroy(frefid_t fref);
```

Destroys a fileref which you have created. This does *not* affect the disk file; it just reclaims the resources allocated by the glk_fileref_create... function.

It is legal to destroy a fileref after opening a file with it (while the file is still open.) The fileref is only used for the opening operation, not for accessing the file stream.

```
frefid_t glk_fileref_iterate(frefid_t fref, glui32 *rockptr);
```

This iterates through all the existing filerefs. See section 1.6.2, "Iterating Through Opaque Objects".

```
glui32 glk_fileref_get_rock(frefid_t fref);
```

This retrieves the fileref's rock value. See section 1.6.1, "Rocks".

```
void glk_fileref_delete_file(frefid_t fref);
```

This deletes the file referred to by fref. It does not destroy the fileref itself.

You should only call this with a fileref that refers to an existing file.

```
glui32 glk_fileref_does_file_exist(frefid_t fref);
```

This returns TRUE (1) if the fileref refers to an existing file, and FALSE (0) if not.

7. Graphics

In accordance with this modern age, Glk provides for a modicum of graphical flair. It does *not* attempt to be a complete graphical toolkit. Those already exist. Glk strikes the usual uncomfortable balance between power, portability, and ease of implementation: commands for arranging pre-supplied images on the screen and intermixed with text.

Graphics is an optional capability in Glk; not all libraries support graphics. This should not be a surprise.

7.1. Image Resources

Most of the graphics commands in Glk deal with image resources. Your program does not have to worry about how images are stored. Everything is a resource, and a resource is referred to by an integer identifier. You may, for example, call a function to display image number 17. The format, loading, and displaying of that image is entirely up to the Glk library for the platform in question.

Of course, it is also desirable to have a platform-independent way to store sounds and images.

Blorb is the official resource-storage format of Glk. A Glk library does not have to understand Blorb, but it is more likely to understand Blorb than any other format.

[Glk does not specify the exact format of images, but Blorb does. Images in a Blorb archive must be PNG or JPEG files. More formats may be added if real-world experience shows it to be desirable. However, that is in the domain of the Blorb specification. The Glk spec, and Glk programming, will not change.]

At present, images can only be drawn in graphics windows and text buffer windows. In fact, a library may not implement both of these possibilities. You should test each with the gestalt_DrawImage selector if you plan to use it. See section 7.4, "Testing for Graphics Capabilities".

```
   glui32 glk_image_get_info(glui32 image, glui32 *width, glui32 *height);
```

This gets information about the image resource with the given identifier. It returns TRUE (1) if there is such an image, and FALSE (0) if not. You can also pass pointers to width and height variables; if the image exists, the variables will be filled in with the width and height of the image, in pixels. (You can pass NULL for either width or height if you don’t care about that information.)

[You should always use this function to measure the size of images when you are creating your display. Do this even if you created the images, and you know how big they "should" be. This is because images may be scaled in translating from one platform to another, or even from one machine to another. A Glk library might display all images larger than their original size, because of screen resolution or player preference. Images will be scaled proportionally, but you still need to call glk_image_get_info() to determine their absolute size.]

```
   glui32 glk_image_draw(winid_t win, glui32 image, glsi32 val1, glsi32 val2);
```

This draws the given image resource in the given window. The position of the image is given by val1 and val2, but their meaning varies depending on what kind of window you are drawing in. See section 7.2, "Graphics in Graphics Windows" and section 7.3, "Graphics in Text Buffer Windows".

This function returns a flag indicating whether the drawing operation succeeded. *[A FALSE result can occur for many reasons. The image data might be corrupted; the library may not have enough memory to operate; there may be no image with the given identifier; the window might not support image display; and so on.]*

```
   glui32 glk_image_draw_scaled(winid_t win, glui32 image, glsi32 val1, glsi32
      val2, glui32 width, glui32 height);
```

This is similar to glk_image_draw(), but it scales the image to the given width and height, instead of using the image’s standard size. (You can measure the standard size with glk_image_get_info().)

If width or height is zero, nothing is drawn. Since those arguments are unsigned integers, they cannot be negative. If you pass in a negative number, it will be interpreted as a very large positive number, which is almost certain to end badly.

7.2. Graphics in Graphics Windows

A graphics window is a rectangular canvas of pixels, upon which you can draw images. The contents are entirely under your control. You can draw as many images as you like, at any positions — overlapping if you like. If the window is resized, you are responsible for redrawing everything. See section 3.5.5, "Graphics Windows".

If you call glk_image_draw() or glk_image_draw_scaled() in a graphics window, val1 and val2 are interpreted as X and Y coordinates. The image will be drawn with its upper left corner at this position.

It is legitimate for part of the image to fall outside the window; the excess is not drawn. Note that these are signed arguments, so you can draw an image which falls outside the left or top edge of the window, as well as the right or bottom.

There are a few other commands which apply to graphics windows:

```
void glk_window_set_background_color(winid_t win, glui32 color);
```

This sets the window's background color. It does *not* change what is currently displayed; it only affects subsequent clears and resizes. The initial background color of each window is white.

Colors are encoded in a 32-bit value: the top 8 bits must be zero, the next 8 bits are the red value, the next 8 bits are the green value, and the bottom 8 bits are the blue value. Color values range from 0 to 255. *[So 0x00000000 is black, 0x00FFFFFF is white, and 0x00FF0000 is bright red.]*

[This function may only be used with graphics windows. To set background colors in a text window, use text styles with color hints; see section 5.5, "Styles".]

```
void glk_window_fill_rect(winid_t win, glui32 color, glsi32 left, glsi32
   top, glui32 width, glui32 height);
```

This fills the given rectangle with the given color. It is legitimate for part of the rectangle to fall outside the window. If width or height is zero, nothing is drawn.

```
void glk_window_erase_rect(winid_t win, glsi32 left, glsi32 top, glui32
   width, glui32 height);
```

This fills the given rectangle with the window's background color.

You can also fill an entire graphics window with its background color by calling glk_window_clear().

[Note that graphics windows do not support a full set of object-drawing commands, nor can you draw text in them. That may be available in a future Glk extension. For now, it seems reasonable to limit the task to a single primitive, the drawing of a raster image. And then there's the ability to fill a rectangle with a solid color — a small extension, and hopefully no additional work for the library, since it can already clear with arbitrary background colors. In fact, if glk_window_fill_rect() did not exist, an author could invent it — by briefly setting the background color, erasing a rectangle, and restoring.]

7.3. Graphics in Text Buffer Windows

A text buffer is a linear text stream. You can draw images in-line with this text. If you are familiar with HTML, you already understand this model. You draw images with flags indicating alignment. The library takes care of scrolling, resizing, and reformatting text buffer windows.

If you call glk_image_draw() or glk_image_draw_scaled() in a text buffer window, val1 gives the image alignment. The val2 argument is currently unused, and should always be zero.

- imagealign_InlineUp: The image appears at the current point in the text, sticking up. That is, the bottom edge of the image is aligned with the baseline of the line of text.
- imagealign_InlineDown: The image appears at the current point, and the top edge is aligned with the top of the line of text.
- imagealign_InlineCenter: The image appears at the current point, and it is centered between

the top and baseline of the line of text. If the image is taller than the line of text, it will stick up and down equally.
- imagealign_MarginLeft: The image appears in the left margin. Subsequent text will be displayed to the right of the image, and will flow around it — that is, it will be left-indented for as many lines as it takes to pass the image.
- imagealign_MarginRight: The image appears in the right margin, and subsequent text will flow around it on the left.

The two "margin" alignments require some care. To allow proper positioning, images using imagealign_MarginLeft and imagealign_MarginRight *must* be placed at the beginning of a line. That is, you may only call glk_image_draw() (with these two alignments) in a window, if you have just printed a newline to the window's stream, or if the window is entirely empty. If you margin-align an image in a line where text has already appeared, no image will appear at all.

Inline-aligned images count as "text" for the purpose of this rule.

You may have images in both margins at the same time.

It is also legal to have more than one image in the *same* margin (left or right.) However, this is not recommended. It is difficult to predict how text will wrap in that situation, and libraries may err on the side of conservatism.

You may wish to "break" the stream of text down below the current margin image. Since lines of text can be in any font and size, you cannot do this by counting newlines. Instead, use this function:

```
void glk_window_flow_break(winid_t win);
```

If the current point in the text is indented around a margin-aligned image, this acts like the correct number of newlines to start a new line below the image. (If there are several margin-aligned images, it goes below all of them.) If the current point is *not* beside a margin-aligned image, this call has no effect.

When a text buffer window is resized, a flow-break behaves cleverly; it may become active or inactive as necessary. You can consider this function to insert an invisible mark in the text stream. The mark works out how many newlines it needs to be whenever the text is formatted for display.

An example of the use of glk_window_flow_break(): If you display a left-margin image at the start of every line, they can stack up in a strange diagonal way that eventually squeezes all the text off the screen. *[If you can't picture this, draw some diagrams. Make the margin images more than one line tall, so that each line starts already indented around the last image.]* To avoid this problem, call glk_window_flow_break() immediately before glk_image_draw() for every margin-aligned image.

In all windows other than text buffers, glk_window_flow_break() has no effect.

7.4. Testing for Graphics Capabilities

Before calling Glk graphics functions, you should use the following gestalt selectors.

```
glui32 res;
res = glk_gestalt(gestalt_Graphics, 0);
```

This returns 1 if the overall suite of graphics functions is available. This includes glk_image_draw(), glk_image_draw_scaled(), glk_image_get_info(), glk_window_erase_rect(), glk_window_fill_rect(), glk_window_set_background_color(), and glk_window_flow_break(). It also includes the capability to create graphics windows.

If this selector returns 0, you should not try to call these functions. They may have no effect, or they may cause a run-time error. If you try to create a graphics window, you will get NULL.

If you are writing a C program, there is an additional complication. A library which does not support graphics may not implement the graphics functions at all. Even if you put gestalt tests around your graphics calls, you may get link-time errors. If the glk.h file is so old that it does not declare the graphics functions and constants, you may even get compile-time errors.

To avoid this, you can perform a preprocessor test for the existence of GLK_MODULE_IMAGE. If this is defined, so are all the functions and constants described in this section. If not, not.

[To be extremely specific, there are two ways this can happen. If the glk.h file that comes with the library is too old to have the graphics declarations in it, it will of course lack GLK_MODULE_IMAGE as well. If the glk.h file is recent, but the library is old, the definition of GLK_MODULE_IMAGE should be removed from glk.h, to avoid link errors. This is not a great solution. A better one is for the library to implement the graphics functions as stubs that do nothing (or cause run-time errors). Since no program will call the stubs without testing gestalt_Graphics, this is sufficient.]

```
   res = glk_gestalt(gestalt_DrawImage, windowtype);
```

This returns 1 if images can be drawn in windows of the given type. If it returns 0, glk_image_draw() will fail and return FALSE (0). You should test wintype_Graphics and wintype_TextBuffer separately, since libraries may implement both, neither, or only one.

```
   res = glk_gestalt(gestalt_GraphicsTransparency, 0);
```

This returns 1 if images with alpha channels can actually be drawn with the appropriate degree of transparency. If it returns 0, the alpha channel is ignored; fully transparent areas will be drawn in an implementation-defined color. *[The JPEG format does not support transparency or alpha channels; the PNG format does.]*

8. Sound

As with graphics, so with sound. Sounds, however, obviously don’t appear in windows. To play a sound in Glk, you must first create a sound channel to hold it. This is an entirely new class of opaque objects; there are create and destroy and iterate and get_rock functions for channels, just as there are for windows and streams and filerefs.

A channel can be playing exactly one sound at a time. If you want to play more than one sound simultaneously, you need more than one sound channel. On the other hand, a single sound can be played on several channels at the same time, or overlapping itself.

Sound is an optional capability in Glk.

8.1. Sound Resources

As with images, sounds are kept in resources, and your program does not have to worry about the formatting or storage. A resource is referred to by an integer identifier.

A resource can theoretically contain any kind of sound data, of any length. A resource can even be infinitely long. *[This would be represented by some sound encoding with a built-in repeat-forever flag — but that is among the details which are hidden from you.]* A resource can also contain two or more channels of sound (stereo data). Do not confuse such in-sound channels with Glk sound channels. A single Glk sound channel suffices to play any sound, even stereo sounds.

[Again, Blorb is the official resource-storage format of Glk. Sounds in Blorb files can be encoded as Ogg, AIFF, or MOD. See the Blorb specification for details.]

8.2. Creating and Destroying Sound Channels

```
schanid_t glk_schannel_create(glui32 rock);
schanid_t glk_schannel_create_ext(glui32 rock, glui32 volume);
```

This creates a sound channel, about as you’d expect.

Remember that it is possible that the library will be unable to create a new channel, in which case glk_schannel_create() will return NULL.

When you create a channel using glk_schannel_create(), it has full volume, represented by the value 0x10000. Half volume would be 0x8000, three-quarters volume would be 0xC000, and so on. A volume of zero represents silence. The glk_schannel_create_ext() call lets you create a channel with the volume already set at a given level.

You can overdrive the volume of a channel by setting a volume greater than 0x10000. However, this is not recommended; the library may be unable to increase the volume past full, or the sound may become distorted. You should always create sound resources with the maximum volume you will need, and then reduce the volume when appropriate using the channel-volume calls.

[Mathematically, these volume changes should be taken as linear multiplication of a waveform represented as linear samples. As I understand it, linear PCM encodes the sound pressure, and therefore a volume of 0x8000 should represent a 6 dB drop.]

Not all libraries support glk_schannel_create_ext(). You should test the gestalt_Sound2 selector before you rely on it; see section 8.5, "Testing for Sound Capabilities".

```
void glk_schannel_destroy(schanid_t chan);
```

Destroy the channel. If the channel is playing a sound, the sound stops immediately (with no notification event).

8.3. Playing Sounds

```
glui32 glk_schannel_play(schanid_t chan, glui32 snd)
```

Begin playing the given sound on the channel. If the channel was already playing a sound (even the same one), the old sound is stopped (with no notification event).

This returns 1 if the sound actually started playing, and 0 if there was any problem. *[The most* *obvious problem is if there is no sound resource with the given identifier. But other problems can occur. For example, the MOD-playing facility in a library might be unable to handle two MODs at the same time, in which case playing a MOD resource would fail if one was already playing.]*

```
glui32 glk_schannel_play_ext(schanid_t chan, glui32 snd, glui32 repeats,
   glui32 notify);
```

This works the same as glk_schannel_play(), but lets you specify additional options. glk_schannel_play(chan, snd) is exactly equivalent to glk_schannel_play_ext(chan, snd, 1, 0).

The repeats value is the number of times the sound should be repeated. A repeat value of -1 (or rather 0xFFFFFFFF) means that the sound should repeat forever. A repeat value of 0 means that the sound will not be played at all; nothing happens. (Although a previous sound on the channel will be stopped, and the function will return 1.)

The notify value should be nonzero in order to request a sound notification event. If you do this, when the sound is completed, you will get an event with type evtype_SoundNotify. The window will be NULL, val1 will be the sound's resource id, and val2 will be the nonzero value you passed as notify.

If you request sound notification, and the repeat value is greater than one, you will get the event only after the *last* repetition. If the repeat value is 0 or -1, you will never get a notification event at all. Similarly, if the sound is stopped or interrupted, or if the channel is destroyed while the sound is playing, there will be no notification event.

Not all libraries support sound notification. You should test the gestalt_Sound2 selector before you rely on it; see section 8.5, "Testing for Sound Capabilities".

Note that you can play a sound on a channel whose volume is zero. This has no audible result, unless you later change the volume; but it produces notifications as usual. You can also play a sound on a paused channel; the sound is paused immediately, and does not progress.

```
glui32 glk_schannel_play_multi(schanid_t *chanarray, glui32 chancount,
   glui32 *sndarray, glui32 soundcount, glui32 notify);
```

This works the same as glk_schannel_play_ext(), except that you can specify more than one sound. The channel references and sound resource numbers are given as two arrays, which must be the same length. The notify argument applies to all the sounds; the repeats value for all the sounds is 1.

All the sounds will begin at exactly the same time.

This returns the number of sounds that began playing correctly. (This will be a number from 0 to soundcount.)

[If the notify argument is nonzero, you will get a separate sound notification event as each sound finishes. They will all have the same val2 value.]

[Note that you have to supply chancount and soundcount as separate arguments, even though they are required to be the same. This is an awkward consequence of the way array arguments are dispatched in Glulx.]

```
void glk_schannel_stop(schanid_t chan);
```

Stops any sound playing in the channel. No notification event is generated, even if you requested one. If no sound is playing, this has no effect.

```
   void glk_schannel_pause(schanid_t chan);
```

Pause any sound playing in the channel. This does not generate any notification events. If the channel is already paused, this does nothing.

New sounds started in a paused channel are paused immediately.

A volume change in progress is *not* paused, and may proceed to completion, generating a notification if appropriate.

```
   void glk_schannel_unpause(schanid_t chan);
```

Unpause the channel. Any paused sounds begin playing where they left off. If the channel is not already paused, this does nothing.

[This means, for example, that you can pause a channel that is currently not playing any sounds. If you then add a sound to the channel, it will not start playing; it will be paused at its beginning. If you later unpause the channel, the sound will commence.]

```
   void glk_schannel_set_volume(schanid_t chan, glui32 vol);
   void glk_schannel_set_volume_ext(schanid_t chan, glui32 vol, glui32
      duration, glui32 notify);
```

Sets the volume in the channel, from 0 (silence) to 0x10000 (full volume). Again, you can overdrive the volume by setting a value greater than 0x10000, but this is not recommended.

If the duration is zero, the change is immediate. Otherwise, the change begins immediately, and occurs smoothly over the next duration milliseconds.

The notify value should be nonzero in order to request a volume notification event. If you do this, when the volume change is completed, you will get an event with type evtype_VolumeNotify. The window will be NULL, val1 will be zero, and val2 will be the nonzero value you passed as notify.

The glk_schannel_set_volume() does not include duration and notify values. Both are assumed to be zero: immediate change, no notification.

You can call these functions between sounds, or while a sound is playing. However, a zero-duration change while a sound is playing may produce unpleasant clicks.

At most one volume change can be occurring on a sound channel at any time. If you call one of these functions while a previous volume change is in progress, the previous change is interrupted. The beginning point of the new volume change should be wherever the previous volume change was interrupted (rather than the previous change’s beginning or ending point).

Not all libraries support thse functions. You should test the appropriate gestalt selectors before you rely on them; see section 8.5, "Testing for Sound Capabilities".

```
   void glk_sound_load_hint(glui32 snd, glui32 flag);
```

This gives the library a hint about whether the given sound should be loaded or not. If the flag is nonzero, the library may preload the sound or do other initialization, so that glk_schannel_play() will be faster. If the flag is zero, the library may release memory or other resources associated with the sound. Calling this function is always optional, and it has no effect on what the library actually plays.

8.4. Other Sound Channel Functions

```
   schanid_t glk_schannel_iterate(schanid_t chan, glui32 *rockptr);
```

This function can be used to iterate through the list of all open channels. See section 1.6.2, "Iterating Through Opaque Objects".

As that section describes, the order in which channels are returned is arbitrary.

```
   glui32 glk_schannel_get_rock(schanid_t chan);
```

This retrieves the channel’s rock value. See section 1.6.1, "Rocks".

8.5. Testing for Sound Capabilities

Before calling Glk sound functions, you should use the following gestalt selectors.

```
   glui32 res;
   res = glk_gestalt(gestalt_Sound2, 0);
```

This returns 1 if the overall suite of sound functions is available. This includes all the functions defined in this chapter. It also includes the capabilities described below under gestalt_SoundMusic, gestalt_SoundVolume, and gestalt_SoundNotify.

If you are writing a C program, there is an additional complication. A library which does not support sound may not implement the sound functions at all. Even if you put gestalt tests around your sound calls, you may get link-time errors. If the glk.h file is so old that it does not declare the sound functions and constants, you may even get compile-time errors.

To avoid this, you can perform a preprocessor test for the existence of GLK_MODULE_SOUND2. If this is defined, so are all the functions and constants described in this section. If not, not.

Earlier versions of the Glk spec defined separate selectors for various optional capabilities. This has proven to be an unnecessarily confusing strategy, and is no longer used. The following selectors still exist, but you should not need to test them; the gestalt_Sound2 selector covers all of them.

```
   res = glk_gestalt(gestalt_Sound, 0);
```

This returns 1 if the overall suite of sound functions is available. This includes glk_schannel_create(), glk_schannel_destroy(), glk_schannel_iterate(), glk_schannel_get_rock(), glk_schannel_play(), glk_schannel_play_ext(), glk_schannel_stop(), glk_schannel_set_volume(), and glk_sound_load_hint().

If this selector returns 0, you should not try to call these functions. They may have no effect, or they may cause a run-time error.

This selector is guaranteed to return 1 if gestalt_Sound2 does.

You can perform a preprocessor test for the existence of GLK_MODULE_SOUND. If this is defined, so are the functions listed above.

```
   res = glk_gestalt(gestalt_SoundMusic, 0);
```

This returns 1 if the library is capable of playing music sound resources. If it returns 0, only sampled sounds can be played.*["Music sound resources" means MOD songs — the only music format that Blorb currently supports. The presence of this selector is, of course, an ugly hack. It is a concession to the current state of the Glk libraries, some of which can handle AIFF but not MOD sounds.]*

This selector is guaranteed to return 1 if gestalt_Sound2 does.

```
   res = glk_gestalt(gestalt_SoundVolume, 0);
```

This selector returns 1 if the glk_schannel_set_volume() function works. If it returns zero, glk_schannel_set_volume() has no effect.

This selector is guaranteed to return 1 if gestalt_Sound2 does.

```
   res = glk_gestalt(gestalt_SoundNotify, 0);
```

This selector returns 1 if the library supports sound notification events. If it returns zero, you will never get such events.

This selector is guaranteed to return 1 if gestalt_Sound2 does.

9. Hyperlinks

Some games may wish to mark up the text in their windows with hyperlinks, which can be selected by the player — most likely by mouse click. Glk allows this in a manner similar to the way text styles are set.

Hyperlinks are an optional capability in Glk.

9.1. Creating Hyperlinks

```
   void glk_set_hyperlink(glui32 linkval);
   void glk_set_hyperlink_stream(strid_t str, glui32 linkval);
```

These calls set the current link value in the current output stream, or the specified output stream, respectively. A link value is any non-zero integer; zero indicates no link. Subsequent text output is considered to make up the body of the link, which continues until the link value is changed (or set to zero).

Note that it is almost certainly useless to change the link value of a stream twice with no intervening text. The result will be a zero-length link, which the player probably cannot see or select; the library may optimize it out entirely.

Setting the link value of a stream to the value it already has, has no effect.

If the library supports images, they take on the current link value as they are output, just as text does. The player can select an image in a link just as he does text. (This includes margin-aligned images, which can lead to some peculiar situations, since a right-margin image may not appear directly adjacent to the text it was output with.)

The library will attempt to display links in some distinctive way (and it will do this whether or not hyperlink input has actually been requested for the window). Naturally, blue underlined text is most likely. Link images may not be distinguished from non-link images, so it is best not to use a particular image both ways.

9.2. Accepting Hyperlink Events

```
void glk_request_hyperlink_event(winid_t win);
void glk_cancel_hyperlink_event(winid_t win);
```

These calls works like the other event request calls. A pending request on a window remains pending until the player selects a link, or the request is cancelled.

A window can have hyperlink input and mouse, character, or line input pending at the same time. However, if hyperlink and mouse input are requested at the same time, the library may not provide an intuitive way for the player to distingish which a mouse click represents. Therefore, this combination should be avoided.

When a link is selected in a window with a pending request, glk_select() will return an event of type evtype_Hyperlink. In the event structure, win tells what window the event came from, and val1 gives the (non-zero) link value.

If no hyperlink request is pending in a window, the library will ignore attempts to select a link. No evtype_Hyperlink event will be generated unless it has been requested.

9.3. Testing for Hyperlink Capabilities

Before calling Glk hyperlink functions, you should use the following gestalt selectors.

```
glui32 res;
res = glk_gestalt(gestalt_Hyperlinks, 0);
```

This returns 1 if the overall suite of hyperlinks functions is available. This includes glk_set_hyperlink(), glk_set_hyperlink_stream(), glk_request_hyperlink_event(), glk_cancel_hyperlink_event().

If this selector returns 0, you should not try to call these functions. They may have no effect, or they may cause a run-time error.

You can test whether hyperlinks are supported with the gestalt_HyperlinkInput selector.

```
res = glk_gestalt(gestalt_HyperlinkInput, windowtype);
```

This will return TRUE (1) if windows of the given type support hyperlinks. If this returns FALSE (0), it is still legal to call glk_set_hyperlink() and glk_request_hyperlink_event(), but they will have no effect, and you will never get hyperlink events.

If you are writing a C program, you can perform a preprocessor test for the existence of GLK_MODULE_HYPERLINKS. If this is defined, so are all the functions and constants described in this section. If not, not.

10. The System Clock

You can get the current time, either as a Unix timestamp (seconds since 1970) or as a broken-out structure of time elements (year, month, day, hour, minute, second).

The system clock is not guaranteed to line up with timer events (see section 4.4, "Timer Events"). Timer events may be delivered late according to the system clock.

```
void glk_current_time(glktimeval_t *time);


typedef struct glktimeval_struct {
    glsi32 high_sec;
    glui32 low_sec;
    glsi32 microsec;
} glktimeval_t;
```

The current Unix time is stored in the structure. (The argument may not be NULL.) This is the number of seconds since the beginning of 1970 (UTC).

The first two values in the structure should be considered a single *signed* 64-bit number. This allows the glktimeval_t to store a reasonable range of values in the future and past. The high_sec value will remain zero until sometime in 2106. If your computer is running in 1969, perhaps due to an unexpected solar flare, then high_sec will be negative.

The third value in the structure represents a fraction of a second, in microseconds (from 0 to 999999). The resolution of the glk_current_time() call is platform-dependent; the microsec value may not be updated continuously.

```
glsi32 glk_current_simple_time(glui32 factor);
```

If dealing with 64-bit values is awkward, you can also get the current time as a lower-resolution 32-bit value. This is simply the Unix time divided by the factor argument (which must not be zero). For example, if factor is 60, the result will be the number of minutes since 1970 (rounded towards negative infinity). If factor is 1, you will get the Unix time directly, but the value will be truncated starting some time in 2038.

10.1. Time and Date Conversions

```
void glk_time_to_date_utc(glktimeval_t *time, glkdate_t *date);
void glk_time_to_date_local(glktimeval_t *time, glkdate_t *date);


typedef struct glkdate_struct {
    glsi32 year;     /* full (four-digit) year */
    glsi32 month;    /* 1-12, 1 is January */
    glsi32 day;      /* 1-31 */
    glsi32 weekday;  /* 0-6, 0 is Sunday */
    glsi32 hour;     /* 0-23 */
```

```
        glsi32 minute;   /* 0-59 */
        glsi32 second;   /* 0-59, maybe 60 during a leap second */
        glsi32 microsec; /* 0-999999 */
    } glkdate_t;
```

Convert the given timestamp (as returned by glk_current_time()) to a broken-out structure. The "utc" function returns a date and time in universal time (GMT); the "local" function returns local time.

[The seconds value may be 60 because of a leap second.]

```
    void glk_simple_time_to_date_utc(glsi32 time, glui32 factor, glkdate_t
       *date);
    void glk_simple_time_to_date_local(glsi32 time, glui32 factor, glkdate_t
       *date);
```

Convert the given timestamp (as returned by glk_current_simple_time()) to a broken-out structure in universal or local time. The time argument is multiplied by factor to produce a Unix timestamp.

Since the resolution of these functions is no better than seconds, they will return zero for the microseconds value.

```
    void glk_date_to_time_utc(glkdate_t *date, glktimeval_t *time);
    void glk_date_to_time_local(glkdate_t *date, glktimeval_t *time);
```

Convert the broken-out structure (interpreted as universal or local time) to a timestamp. The weekday value in glkdate_t is ignored. The other values need not be in their normal ranges; they will be normalized.

If the time cannot be represented by the platform’s time library, this may return -1 for the seconds value. (I.e., the high_sec and low_sec fields both $FFFFFFFF. The microseconds field is undefined in this case.)

The glk_date_to_time_local() function may not be smart about Daylight Saving Time conversions. *[If implemented with the mktime() libc function, it should use the negative tm_isdst flag to "attempt to divine whether summer time is in effect".]*

```
    glsi32 glk_date_to_simple_time_utc(glkdate_t *date, glui32 factor);
    glsi32 glk_date_to_simple_time_local(glkdate_t *date, glui32 factor);
```

Convert the broken-out structure (interpreted as universal or local time) to a timestamp divided by factor. The weekday value in glkdate_t is ignored. The other values need not be in their normal ranges; they will be normalized.

If the time cannot be represented by the platform’s time library, this may return -1.

10.2. Testing for Clock Capabilities

Before calling Glk date and time functions, you should use the following gestalt selector.

```
    res = glk_gestalt(gestalt_DateTime, 0);
```

This returns 1 if the overall suite of system clock functions, as described in this chapter, is available.

If this selector returns 0, you should not try to call these functions. They may have no effect, or they may cause a run-time error.

[Glk timer events are covered by a different selector. See section 4.4, "Timer Events").]

If you are writing a C program, you can perform a preprocessor test for the existence of GLK_MODULE_DATETIME. If this is defined, so are all the functions and data types described in this section.

11. Porting, Adapting, and Other Messy Bits

Life is not perfect, and neither are our toys. In a world of perfect toys, a Glk program could compile with any Glk library and run without human intervention. Guess what.

11.1. Startup Options

One large grey area is starting up, startup files, and other program options. It is easy to assume that all C programs run with the (argc, argv) model — that all the information they need comes as an array of strings at startup time. This is sometimes true. But in a GUI system, files are often opened by clicking, options are specified in dialog boxes, and so on; and this does not necessarily happen at the beginning of main().

Therefore, Glk does not try to pass an (argc, argv) list to your glk_main(). Nor does it provide a portable API for startup files and options. *[Doing that well would require API calls to parse command-line arguments of various types, and then also design and handle dialog boxes. It would go far beyond the level of complexity which Glk aspires to.]*

Instead, startup files and options are handled in an *entirely platform-dependent* manner. You, as the author of a Glk program, must describe how your program should behave. As your program is ported to various Glk libraries, each porter must decide how to implement that behavior on the platform in question. The library should store the options and files in global variables, where your glk_main() routine can read them.

It is reasonable to modularize this code, and call it the "startup code". But the startup code is not necessarily a single function, and it certainly does not have well-defined arguments such as an (argc, argv) list. You can consider that your startup behavior is divided into the messy part, which is nonportable and goes in the startup code, and the clean part, which is entirely Glk-portable and goes at the beginning of glk_main().

This is not as much of a mess as it sounds. Many programs, and almost all IF programs, follow one of a few simple models.

- The simple model: There are no startup files. The program just starts running when invoked.
- The game-file model: The program begins running when it is handed a single file of a particular type. On command-line systems, this comes as a filename in a command-line option. On GUI systems, it will usually be a platform-native event which contains a file reference.

Any Glk library will be able to support these two models, probably through compile-time options. The details will vary. *[For one notable case, the Mac Glk library has two possible behaviors when* *compiled with the game-file model. If the player double-clicks a game file, the library calls glk_main() immediately. If the player double-clicks the application icon, the library allows the player to wait, perhaps adjusting preferences; it only calls glk_main() after the game file is selected through a file dialog.]*

[In fact, if life were this simple, it would be worth adding these models to the Glk API somehow. Unfortunately, it's not. Consider AGT: the "game file" is actually about ten separate files with related filenames, in the same directory. Glk does not contain API calls to do precise file and pathname manipulation; it is too complicated an area to support. So this situation must be handled non-portably.]

More complicated models are also possible. You might want to accept files through GUI events at any time, not just at startup. This could be handled by defining a new Glk event type, and having the library send such an event when a platform-native icon-click is detected. You would then have to decide how the situation should be handled in a command-line Glk library. But that is inherent in your task as a program author.

Options and preferences are a separate problem. Most often, a command-line library will handle them with command-line arguments, and a GUI library will handle them with a dialog box. Ideally, you should describe how both cases should behave — list the command-line arguments, and perhaps how they could be labelled in a dialog. *[This is unlikely to be very complicated. Although who knows.]*

Remember that the Glk library is likely to have some options of its own — matters of display styles and so on. A command-line library will probably have a simple API to extract its own options and pass the rest on to the startup code.

11.2. Going Outside the Glk API

Nonportable problems are not limited to the start of execution. There is also the question of OS services which are not represented in Glk. The ANSI C libraries are so familiar that they seem universal, but they are actually not necessarily present. Palmtop platforms such as PalmOS are particularly good at leaving out ANSI libraries.

11.2.1. Memory Management

Everyone uses malloc(), realloc(), and free(). However, some platforms have a native memory-management API which may be more suitable in porting your program.

The malloc() system is simple; it can probably be implemented as a layer on top of whatever native API is available. So you don't absolutely have to worry about this. However, it can't hurt to group all your malloc() and free() calls in one part of your program, so that a porter can easily change them all if it turns out to be a good idea.

11.2.2. String Manipulation

This is more of a nuisance, because the set of string functions varies quite a bit between platforms. Consider bcopy(), memcpy(), and memmove(); stricmp() and strcasecmp(); strchr() and index(); and so on. And again, on a palmtop machine, none of these may be available. The maximally safe course is to implement what you need yourself. *[See the model.c program for an example; it implements its own str_eq() and str_len().]*

The maximally safe course is also a pain in the butt, and may well be inefficient (a platform may have a memcpy() which is highly optimized for large moves.) That's porting in the big city.

[By the way, the next person I see who #defines memmove() as memcpy() when a real memmove() isn't available, gets slapped in the face with a lead-lined rubber salmon.]

11.2.3. File Handling

This is the real nuisance, because Glk provides a limited set of stream and file functions. And yet there are all these beautiful ANSI stdio calls, which have all these clever tricks — ungetc(), fast macro fgetc(), formatted fprintf(), not to mention the joys of direct pathname manipulation. Why bother with the Glk calls?

The problem is, the stdio library really isn't always the best choice, particularly on mobile OSes.

There's also the problem of hooking into the Glk API. Window output goes through Glk streams. *[It would have been lovely to use the stdio API for that, but it's not generally possible.]*

As usual, it's a judgement call. If you have a large existing pile of source code which you're porting, and it uses a lot of icky stdio features like ungetc(), it may be better not to bother changing everything to the Glk file API. If you're starting from scratch, using the Glk calls will probably be cleaner.

11.2.4. Private Extensions to Glk

Sometimes — hopefully rarely — there's stuff you just gotta do.

Explicit pathname modification is one possible case. Creating or deleting directories. New Glk event types caused by interface events. Control over pull-down menus.

Like startup code, you just have to decide what you want, and ask your porters to port it. These are the non-portable parts of your task. As I said, that's porting in the big city.

If an extension or new function is so useful that everyone is implementing it, I'll consider adding it to the Glk API (as an optional capability, with a Gestalt selector and everything.) I'm flexible. In a morally correct manner, of course.

11.3. Glk and the Virtual Machine

Most IF games are built on a virtual machine, such as the Z-machine or the TADS runtime structure. Building a virtual machine which uses Glk as its interface is somewhat more complicated than writing a single Glk program.

The question to ask is: what API will be exported to the game author — the person writing a program to run on the VM?

11.3.1. Implementing a Higher Layer Over Glk

Thus far, each virtual machine has had its own built-in I/O API. Most of them have identical basic capabilities — read lines of input, display a stream of output, show a status line of some sort, and so on. This commonality, of course, is the ground from which Glk sprouted in the first place.

If the I/O API is a subset of the capabilities of Glk, it can be implemented as a layer on top of Glk. In this way, an existing VM can often be ported to Glk without any change visible to the author. Standard TADS can be ported in this way; the V5/8 Z-machine can as well (with the sole exception, as far as I know, of colored text.)

11.3.2. Glk as a VM's Native API

The other approach is to use Glk as the virtual machine's own I/O API, and provide it directly to the game author. The Glulx virtual machine is built this way. This is inherently more powerful, since it allows access to all of Glk, instead of a subset. As Glk is designed to be easily expandable, and will gain new (optional) capabilities over time, this approach also allows a VM to gain capabilities over time without much upheaval.

[To a certain extent, Glk was designed for this use more than any other. For example, this is why all Glk function arguments are either pointers or 32-bit integers, and why all Glk API structures are effectively arrays of same. It is also why the iterator functions exist; a VM's entire memory space may be reset by an "undo" or "restore" command, and it would then have to, ah, take inventory of its streams and windows and filerefs.]

[This is also another reason why Glk provides file API calls. A VM can provide Glk as the game author's entire access to the file system, as well as the author's entire access to the display system. The VM will then be simpler, more modular, not as tied into the native OS — all that good stuff.]

[The Society of C Pedants wishes me to point out that the structures in the Glk API aren't really *arrays of 32-bit integers. A structure made up entirely of 32-bit integers can still be padded weirdly by a C compiler. This problem is solved cleanly by the dispatch layer; see below.]*

The mechanics of this are tricky, because Glk has many API calls, and more will be added over time.

In a VM with a limited number of opcodes, it may be best to allocate a single "Glk" opcode, with a variable number of arguments, the first of which is a function selector. (Glulx does this.) Allow at least 16 bits for this selector; there may be more than 256 Glk calls someday. (For a list of standard selectors for Glk calls, see section 12.1.6, "Table of Selectors".)

In a VM with a large opcode space, you could reserve a 16-bit range of opcodes for Glk.

It may also be feasible to extend the function-call mechanism in some way, to include the range of Glk functions.

In any case, the API still has to be exported to the game author in whatever language is compiled to the VM. Ideally, this can be done as a set of function calls. *[But it doesn't have to be. The Inform compiler, for example, can accept assembly opcodes in-line with Inform source code. It's nearly as convenient to let the author type in opcodes as function calls.]*

There is a further complication when new calls are added to Glk. This should not be a major problem. The compiler is mapping Glk calls one-to-one to its own functions or opcodes, so this should be a matter of adding to a fixed list somewhere in the compiler and releasing an upgrade.

Alternatively, if the compiler has some way to define new opcodes, even this much effort is not necessary. *[The Inform compiler is designed this way; the game author can define new opcodes and use them. So if a new call has been added to Glk, and it has been implemented in the interpreter with a known selector, it can be used in Inform immediately, without a compiler upgrade.]*

Or, you can provide a completely dynamic interface to the Glk API. This is the province of the Glk dispatch layer, which is not part of Glk proper; it rests on top. See section 12.1, "The Dispatch Layer".

12. Appendices

12.1. The Dispatch Layer

The material described in this section is not part of the Glk API per se. It is an external layer, lying on top of Glk, which allows a program to invoke Glk dynamically — determining the capabilities and interfaces of Glk at run-time.

This is most useful for virtual machines and other run-time systems, which want to use Glk without being bound to a particular version of the Glk API. In other words, a VM can export Glk to VM programs, without hard-wiring a list of Glk functions within itself. If a new Glk library is released, with new functions, the VM can simply link in the library; the new functions will be available to VM programs without further work.

If you are writing a C program which uses the Glk API, you can ignore this section entirely. If you are writing a VM which uses Glk, you need to read it. If you are implementing a Glk library, you should also read it. (There are some additional interfaces which your library must support for the dispatch layer to work right.)

12.1.1. How This Works

The dispatch layer is implemented in a C source file, gi_dispa.c, and its header, gi_dispa.h. This code is platform-independent — it is identical in every library, just as the glk.h header file is identical in every library. Each library author should download the gi_dispa.c and gi_dispa.h files from the Glk web site, and compile them unchanged into the library.

This code is mostly external to Glk; it operates by calling the documented Glk API, not library internals. This is how gi_dispa.c can be platform-independent. However, the dividing line is not perfect. There are a few extra functions, not part of the Glk API, which the library must implement; gi_dispa.c (and no one else) calls these functions. These functions are simple and should not make life much harder for library implementors.

The dispatch layer then provides a dispatch API. The heart of this is the gidispatch_call() function, which allows you to call *any* Glk function (specified by number) and pass in a list of arguments in a standardized way. You may also make use of gidispatch_prototype(), which gives you the proper format of that list for each function. Ancilliary functions let you enumerate the functions and constants in the Glk API.

12.1.2. Interrogating the Interface

These are the ancilliary functions that let you enumerate.

```
glui32 gidispatch_count_classes(void);
```

This returns the number of opaque object classes used by the library. You will need to know this if you want to keep track of opaque objects as they are created; see section 12.1.5.1, "Opaque Object Registry".

As of Glk API 0.6.0, there are four classes: windows, streams, filerefs, and sound channels (numbered 0, 1, 2, and 3 respectively.)

```
glui32 gidispatch_count_intconst(void);
```

This returns the number of integer constants exported by the library.

```
gidispatch_intconst_t *gidispatch_get_intconst(glui32 index);


typedef struct gidispatch_intconst_struct {
    char *name;
    glui32 val;
} gidispatch_intconst_t;
```

This returns a structure describing an integer constant which the library exports. These are, roughly, all the constants defined in the glk.h file. index can range from 0 to N-1, where N is the value returned by gidispatch_count_intconst().

The structure simply contains a string and a value. The string is a symbolic name of the value, and can be re-exported to anyone interested in using Glk constants.

[In the current gi_dispa.c library, these structures are static and immutable, and will never be deallocated. However, it is safer to assume that the structure may be reused in future gidispatch_get_intconst() calls.]

```
    glui32 gidispatch_count_functions(void);
```

This returns the number of functions exported by the library.

```
    gidispatch_function_t *gidispatch_get_function(glui32 index);


    typedef struct gidispatch_function_struct {
        glui32 id;
        void *fnptr;
        char *name;
    } gidispatch_function_t;
```

This returns a structure describing a Glk function. index can range from 0 to N-1, where N is the value returned by gidispatch_count_functions().

The id field is a selector — a numeric constant used to refer to the function in question. name is the function name, as it is given in the glk.h file, but without the "glk_" prefix. And fnptr is the address of the function itself. *[This is included because it might be useful, but it is* not *recommended. To call an arbitrary Glk function, you should use gidispatch_call().]* See section 12.1.6, "Table of Selectors" for the selector definitions. See section 12.1.3, "Dispatching" for more about calling Glk functions by selector.

```
    gidispatch_function_t *gidispatch_get_function_by_id(glui32 id);
```

This returns a structure describing the Glk function with selector id. If there is no such function in the library, this returns NULL.

[Again, it is safest to assume that the structure is only valid until the next gidispatch_get_function() or gidispatch_get_function_by_id() call.]

12.1.3. Dispatching

```
    void gidispatch_call(glui32 funcnum, glui32 numargs, gluniversal_t
       *arglist);
```

funcnum is the function number to invoke; see section 12.1.6, "Table of Selectors". arglist is the list of arguments, and numargs is the length of the list.

The arguments are all stored as gluniversal_t objects. This is a union, encompassing all the types that can be passed to Glk functions.

```
typedef union gluniversal_union {
    glui32 uint;
    glsi32 sint;
    void *opaqueref;
    unsigned char uch;
    signed char sch;
    char ch;
    char *charstr;
    void *array;
    glui32 ptrflag;
} gluniversal_t;
```

12.1.3.1. Basic Types

Numeric arguments are passed in the obvious way — one argument per gluniversal_t, with the uint or sint field set to the numeric value. Characters and strings are also passed in this way — chars in the uch, sch, or ch fields (depending on whether the char is signed) and strings in the charstr field. Opaque objects (windows, streams, etc) are passed in the opaqueref field (which is void*, in order to handle all opaque pointer types.)

However, pointers (other than C strings), arrays, and structures complicate life. So do return values.

12.1.3.2. References

A reference to a numeric type or object reference — that is, glui32*, winid_t*, and so on — takes *one or two* gluniversal_t objects. The first is a flag indicating whether the reference argument is NULL or not. The ptrflag field of this gluniversal_t should be FALSE (0) if the reference is NULL, and TRUE (1) otherwise. If FALSE, that is the end of the argument; you should not use a gluniversal_t to explicitly store the NULL reference. If the flag is TRUE, you must then put a gluniversal_t storing the base type of the reference.

For example, consider a hypothetical function, with selector 0xABCD:

```
void glk_glomp(glui32 num, winid_t win, glui32 *numref, strid_t *strref);
```

...and the calls:

```
glui32 value;
winid_t mainwin;
strid_t gamefile;
glk_glomp(5, mainwin, &value, &gamefile);
```

To perform this through gidispatch_call(), you would do the following:

```
gluniversal_t arglist[6];
arglist[0].uint = 5;
arglist[1].opaqueref = mainwin;
```

```
arglist[2].ptrflag = TRUE;
arglist[3].uint = value;
arglist[4].ptrflag = TRUE;
arglist[5].opaqueref = gamefile;
gidispatch_call(0xABCD, 6, arglist);
value = arglist[3].uint;
gamefile = arglist[5].opaqueref;
```

Note that you copy the value of the reference arguments into and out of arglist. Of course, it may be that glk_glomp() only uses these as pass-out references or pass-in references; if so, you could skip copying in or out.

For further examples:

```
glk_glomp(7, mainwin, NULL, NULL);
...or...
gluniversal_t arglist[4];
arglist[0].uint = 7;
arglist[1].opaqueref = mainwin;
arglist[2].ptrflag = FALSE;
arglist[3].ptrflag = FALSE;
gidispatch_call(0xABCD, 4, arglist);
```

```
glk_glomp(13, NULL, NULL, &gamefile);
...or...
gluniversal_t arglist[5];
arglist[0].uint = 13;
arglist[1].opaqueref = NULL;
arglist[2].ptrflag = FALSE;
arglist[3].ptrflag = TRUE;
arglist[4].opaqueref = gamefile;
gidispatch_call(0xABCD, 5, arglist);
gamefile = arglist[4].opaqueref;
```

```
glk_glomp(17, NULL, &value, NULL);
...or...
gluniversal_t arglist[5];
arglist[0].uint = 17;
arglist[1].opaqueref = NULL;
arglist[2].ptrflag = TRUE;
arglist[3].uint = value;
arglist[4].ptrflag = FALSE;
gidispatch_call(0xABCD, 5, arglist);
value = arglist[3].uint;
```

As you see, the length of arglist depends on how many of the reference arguments are NULL.

12.1.3.3. Structures

A structure pointer is represented by a single ptrflag, possibly followed by a sequence of gluniversal_t objects (one for each field of the structure.) Again, if the structure pointer is non-NULL, the ptrflag should be TRUE and be followed by values; if not, the ptrflag should be NULL and stands alone.

For example, the function glk_select() can be invoked as follows:

```
event_t ev;
gluniversal_t arglist[5];
arglist[0].ptrflag = TRUE;
gidispatch_call(0x00C0, 5, arglist);
ev.type = arglist[1].uint;
ev.win = arglist[2].opaqueref;
ev.val1 = arglist[3].uint;
ev.val2 = arglist[4].uint;
```

Since the structure passed to glk_select() is a pass-out reference (the entry values are ignored), you don't need to fill in arglist[1..4] before calling gidispatch_call().

[Theoretically, you would invoke glk_select(NULL) by setting arglist[0].ptrflag to FALSE, and using a one-element arglist instead of five-element. But it's illegal to pass NULL to glk_select(). So you cannot actually do this.]

12.1.3.4. Arrays

In the Glk API, an array argument is always followed by a numeric argument giving the array's length. These two C arguments are a single logical argument, which is represented by *one or three* gluniversal_t objects. The first is a ptrflag, indicating whether the argument is NULL or not. The second is a pointer, stored in the array field. The third is the array length, stored in the uint field. And again, if the ptrflag is NULL, the following two are omitted.

For example, the function glk_put_buffer() can be invoked as follows:

```
char buf[64];
glui32 len = 64;
glk_put_buffer(buf, len);
...or...
gluniversal_t arglist[3];
arglist[0].ptrflag = TRUE;
arglist[1].array = buf;
arglist[2].uint = len;
gidispatch_call(0x0084, 3, arglist);
```

Since you are passing a C char array to gidispatch_call(), the contents will be read directly from that. There is no need to copy data into arglist, as you would for a basic type.

If you are implementing a VM whose native representation of char arrays is more complex, you will have to do more work. You should allocate a C char array, copy your characters into it, make the call, and then free the array. *[glk_put_buffer() does not modify the array passed to it, so there is no need to copy the characters out.]*

12.1.3.5. Return Values

The return value of a function is not treated specially. It is simply considered to be a pass-out reference argument which may not be NULL. It comes after all the other arguments of the function.

For example, the function glk_window_get_rock() can be invoked as follows:

```
    glui32 rock;
    winid_t win;
    rock = glk_window_get_rock(win);
    ...or...
    gluniversal_t arglist[3];
    arglist[0].opaqueref = win;
    arglist[1].ptrflag = TRUE;
    gidispatch_call(0x0021, 3, arglist);
    rock = arglist[2].uint;
```

12.1.4. Getting Argument Prototypes

There are many possible ways to set up a gluniversal_t array, and it's illegal to call gidispatch_call() with an array which doesn't match the function. Furthermore, some references are passed in, some passed out, and some both. How do you know how to handle the argument list?

One possibility is to recognize each function selector, and set up the arguments appropriately. However, this entails writing special code for each Glk function; which is exactly what we don't want to do.

Instead, you can call gidispatch_prototype().

char *gidispatch_prototype(glui32 funcnum);

This returns a string which encodes the proper argument list for the given function. If there is no such function in the library, this returns NULL.

The prototype string for the glk_glomp() function described above would be: "4IuQa&Iu&Qb:". The "4" is the number of arguments (including the return value, if there is one, which in this case there isn't.) "Iu" denotes an unsigned integer; "Qa" is an opaque object of class 0 (window). "&Iu" is a *reference* to an unsigned integer, and "&Qb" is a reference to a stream. The colon at the end terminates the argument list; the return value would follow it, if there was one.

Note that the initial number ("4" in this case) is the number of logical arguments, not the number of gluniversal_t objects which will be passed to gidispatch_call(). The glk_glomp() call uses anywhere from four to six gluniversal_t objects, as demonstrated above.

The basic type codes:

- Iu, Is: Unsigned and signed 32-bit integer.
- Cn, Cu, Cs: Character, unsigned char, and signed char. *[Of course Cn will be the same as either Cu or Cs, depending on the platform. For this reason, Glk avoids using it, but it is included here for completeness.]*
- S: A C-style string (null-terminated array of char). In Glk, strings are always treated as read-only and used immediately; the library does not retain a reference to a string between Glk calls. A Glk call that wants to use writable char arrays will use an array type ("#C"), not string ("S").
- U: A zero-terminated array of 32-bit integers. This is primarily intended as a Unicode equivalent of "S". Like "S" strings, "U" strings are read-only and used immediately. A Glk call that wants to use writable Unicode arrays will use an array type ("#Iu") instead of "U".

- F: A floating-point value. Glk does not currently use floating-point values, but we might as well define a code for them.
- Qa, Qb, Qc...: A reference to an opaque object. The second letter determines which class is involved. (The number of classes can be gleaned from gidispatch_count_classes(); see section 12.1.2, "Interrogating the Interface"). *[If Glk expands to have more than 26 classes, we'll think of something.]*

Any type code can be prefixed with one or more of the following characters:

- &: A reference to the type; or, if you like, a variable passed by reference. The reference is passed both in and out, so you must copy the value in before calling gidispatch_call() and copy it out afterward.
- <: A reference which is pass-out only. The initial value is ignored, so you only need copy out the value after the call.
- >: A reference which is pass-in only. *[This is not generally used for simple types, but is useful for structures and arrays.]*
- +: Combined with "&", "<", or ">", indicates that a valid reference is mandatory; NULL cannot be passed. *[Note that even though the ptrflag gluniversal_t for a "+" reference is always TRUE, it cannot be omitted.]*
- :: The colon separates the arguments from the return value, or terminates the string if there is no return value. Since return values are always non-NULL pass-out references, you may treat ":" as equivalent to "<+". The colon is never combined with any other prefix character.
- [...]: Combined with "&", "<", or ">", indicates a structure reference. Between the brackets is a complete argument list encoding string, including the number of arguments. *[For example, the prototype string for glk_select() is "1<+[4IuQaIuIu]:" — one argument, which is a pass-out non-NULL reference to a structure, which contains four arguments.]* Currently, structures in Glk contain only basic types.
- #: Combined with "&", "<", or ">", indicates an array reference. As described above, this encompasses up to three gluniversal_t objects — ptrflag, pointer, and integer length. *[Depending on the design of your program, you may wish to pass a pointer directly to your program's memory, or allocate an array and copy the contents in and out. See section 12.1.3.4, "Arrays".]*
- !: Combined with "#", indicates that the array is *retained* by the library. The library will keep a reference to the array; the contents are undefined until further notice. You should not use or copy the contents of the array out after the call, even for "&#!" or "<#!" arrays. Instead, do it when the library releases the array. *[For example, glk_stream_open_memory() retains the array that you pass it, and releases it when the stream is closed. The library can notify you automatically when arrays are retained and released; see section 12.1.5.2, "Retained Array Registry".]*

The order of these characters and prefixes is not completely arbitrary. Here is a formal grammar for the prototype strings. *[Thanks to Neil Cerutti for working this out.]*

```
<prototype>   ->  ArgCount [ <arg_list> ] ':' [ <arg> ] EOL
<arg_list>    ->  <arg> { <arg> }
<arg>         ->  TypeName | <ref_type>
<ref_type>    ->  RefType [ '+' ] <target_type>
<target_type> ->  TypeName | <array> | <struct>
<array>       ->  '#' [ '!' ] TypeName
<struct>      ->  '[' ArgCount [ <arg_list> ] ']'

TypeName is "I[us]|C[nus]|S|U|F|Q[a-z]"
ArgCount is '\d+'
RefType is '&|<|>'
EOL is end of input
```

12.1.5. Functions the Library Must Provide

Ideally, the three layers — program, dispatch layer, Glk library — would be completely modular; each would refer only to the layers beneath it. Sadly, there are a few places where the library must notify the program that something has happened. Worse, these situations are only relevant to programs which use the dispatch layer, and then only some of those.

Since C is uncomfortable with the concept of calling functions which may not exist, Glk handles this with call-back function pointers. The program can pass callbacks in to the library; if it does, the library will call them, and if not, the library doesn't try.

These callbacks are optional, in the sense that the program may or may not set them. However, any library which wants to interoperate with the dispatch layer must *allow* the program to set them; it is the program's choice. The library does this by implementing set_registry functions — the functions to which the program passes its callbacks.

[Even though these callbacks and the functions to set them are declared in gi_dispa.h, they are not defined in gi_dispa.c. The dispatch layer merely coordinates them. The program defines the callback functions; the library calls them.]

12.1.5.1. Opaque Object Registry

The Glk API refers to opaque objects by pointer; but a VM probably cannot store pointers to native memory. Therefore, a VM program will want to keep a VM-accessible collection of opaque objects. *[For example, it might keep a hash table for each opaque object class, mapping integer identifiers to object pointers.]*

To make this possible, a Glk library must implement gidispatch_set_object_registry().

```
void gidispatch_set_object_registry(gidispatch_rock_t (*reg)(void *obj,
   glui32 objclass), void (*unreg)(void *obj, glui32 objclass,
   gidispatch_rock_t objrock));
```

Your program calls this early (before it begins actually executing VM code.) You pass in two function pointers, matching the following prototypes:

```
gidispatch_rock_t my_vm_reg_object(void *obj, glui32 objclass);
void my_vm_unreg_object(void *obj, glui32 objclass, gidispatch_rock_t
   objrock);
```

Whenever the Glk library creates an object, it will call my_vm_reg_object(). It will pass the object pointer and the class number (from 0 to N-1, where N is the value returned by gidispatch_count_classes().)

You can return any value in the gidispatch_rock_t object; the library will stash this away inside the object. *[Note that this is entirely separate from the regular Glk rock, which is always a glui32 and can be set independently.]*

```
typedef union glk_objrock_union {
    glui32 num;
    void *ptr;
} gidispatch_rock_t;
```

Whenever the Glk library destroys an object, it will call my_vm_unreg_object(). It passes you the object pointer, class number, and the object rock.

You can, at any time, get the object rock of an object. The library implements this function:

```
gidispatch_rock_t gidispatch_get_objrock(void *obj, glui32 objclass);
```

With this and your two callbacks, you can maintain (say) a hash table for each object class, and easily convert back and forth between hash table keys and Glk object pointers. A more sophisticated run-time system (such as Java) could create a typed VM object for every Glk object, thus allowing VM code to manipulate Glk objects intelligently.

One significant detail: It is possible that some Glk objects will already exist when your glk_main() function is called. *[For example, MacGlk can open a stream when the user double-clicks a file; this occurs before glk_main().]* So when you call gidispatch_set_object_registry(), it may *immediately* call your my_vm_reg_object() callback, notifying you of the existing objects. You must be prepared for this possibility. *[If you are keeping hash tables, for example, create them* before *you call gidispatch_set_object_registry().]*

12.1.5.2. Retained Array Registry

A few Glk functions take an array and hold onto it. The memory is "owned" by the library until some future Glk call releases it. While the library retains the array, your program should not read, write, move, or deallocate it. When the library releases it, the contents are in their final form, and you can copy them out (if appropriate) and dispose of the memory as you wish.

To allow this, the library implements gidispatch_set_retained_registry().

```
void gidispatch_set_retained_registry(gidispatch_rock_t (*reg)(void *array,
   glui32 len, char *typecode), void (*unreg)(void *array, glui32 len, char
   *typecode, gidispatch_rock_t objrock));
```

Again, you pass in two function pointers:

```
gidispatch_rock_t my_vm_reg_array(void *array, glui32 len, char *typecode);
void my_vm_unreg_array(void *array, glui32 len, char *typecode,
   gidispatch_rock_t objrock);
```

Whenever a Glk function retains an array, it will call my_vm_reg_array(). This occurs only if you pass an array to an argument with the "#!" prefix. *[But not in every such case. Wait for the my_vm_reg_array() call to confirm it.]* The library passes the array and its length, exactly as you put them in the gluniversal_t array. It also passes the string which describes the argument.

[Currently, the only calls that retain arrays are glk_request_line_event(), glk_stream_open_memory(), glk_request_line_event_uni(), and glk_stream_open_memory_uni(). The first two of these use arrays of characters, so the string is "&+#!Cn". The latter two use arrays of glui32, so the string is "&+#!Iu".]

You can return any value in the gidispatch_rock_t object; the library will stash this away with the array.

When a Glk function releases a retained array, it will call my_vm_unreg_array(). It passes back the same array, len, and typecode parameters, as well as the gidispatch_rock_t you returned from my_vm_reg_array().

With these callbacks, you can maintain a collection of retained arrays. You can use this to copy data from C arrays to your own data structures, or keep relocatable memory locked, or prevent a garbage-collection system from deallocating an array while Glk is writing to it.

12.1.6. Table of Selectors

These values, and the values used for future Glk calls, are integers in the range 0x0001 to 0xFFFF (1 to 65535). The values are not sequential; they are divided into groups, roughly by category. Zero is not the selector of any Glk call, so it may be used for a null value.

- 0x0001: glk_exit
- 0x0002: glk_set_interrupt_handler
- 0x0003: glk_tick
- 0x0004: glk_gestalt
- 0x0005: glk_gestalt_ext
- 0x0020: glk_window_iterate
- 0x0021: glk_window_get_rock
- 0x0022: glk_window_get_root
- 0x0023: glk_window_open
- 0x0024: glk_window_close
- 0x0025: glk_window_get_size
- 0x0026: glk_window_set_arrangement
- 0x0027: glk_window_get_arrangement
- 0x0028: glk_window_get_type
- 0x0029: glk_window_get_parent
- 0x002A: glk_window_clear
- 0x002B: glk_window_move_cursor
- 0x002C: glk_window_get_stream
- 0x002D: glk_window_set_echo_stream
- 0x002E: glk_window_get_echo_stream
- 0x002F: glk_set_window
- 0x0030: glk_window_get_sibling
- 0x0040: glk_stream_iterate
- 0x0041: glk_stream_get_rock
- 0x0042: glk_stream_open_file
- 0x0043: glk_stream_open_memory
- 0x0044: glk_stream_close
- 0x0045: glk_stream_set_position
- 0x0046: glk_stream_get_position
- 0x0047: glk_stream_set_current
- 0x0048: glk_stream_get_current
- 0x0049: glk_stream_open_resource
- 0x0060: glk_fileref_create_temp
- 0x0061: glk_fileref_create_by_name
- 0x0062: glk_fileref_create_by_prompt
- 0x0063: glk_fileref_destroy
- 0x0064: glk_fileref_iterate
- 0x0065: glk_fileref_get_rock
- 0x0066: glk_fileref_delete_file
- 0x0067: glk_fileref_does_file_exist
- 0x0068: glk_fileref_create_from_fileref

- 0x0080: glk_put_char
- 0x0081: glk_put_char_stream
- 0x0082: glk_put_string
- 0x0083: glk_put_string_stream
- 0x0084: glk_put_buffer
- 0x0085: glk_put_buffer_stream
- 0x0086: glk_set_style
- 0x0087: glk_set_style_stream
- 0x0090: glk_get_char_stream
- 0x0091: glk_get_line_stream
- 0x0092: glk_get_buffer_stream
- 0x00A0: glk_char_to_lower
- 0x00A1: glk_char_to_upper
- 0x00B0: glk_stylehint_set
- 0x00B1: glk_stylehint_clear
- 0x00B2: glk_style_distinguish
- 0x00B3: glk_style_measure
- 0x00C0: glk_select
- 0x00C1: glk_select_poll
- 0x00D0: glk_request_line_event
- 0x00D1: glk_cancel_line_event
- 0x00D2: glk_request_char_event
- 0x00D3: glk_cancel_char_event
- 0x00D4: glk_request_mouse_event
- 0x00D5: glk_cancel_mouse_event
- 0x00D6: glk_request_timer_events
- 0x00E0: glk_image_get_info
- 0x00E1: glk_image_draw
- 0x00E2: glk_image_draw_scaled
- 0x00E8: glk_window_flow_break
- 0x00E9: glk_window_erase_rect
- 0x00EA: glk_window_fill_rect
- 0x00EB: glk_window_set_background_color
- 0x00F0: glk_schannel_iterate
- 0x00F1: glk_schannel_get_rock
- 0x00F2: glk_schannel_create
- 0x00F3: glk_schannel_destroy
- 0x00F4: glk_schannel_create_ext
- 0x00F7: glk_schannel_play_multi
- 0x00F8: glk_schannel_play
- 0x00F9: glk_schannel_play_ext
- 0x00FA: glk_schannel_stop
- 0x00FB: glk_schannel_set_volume
- 0x00FC: glk_sound_load_hint
- 0x00FD: glk_schannel_set_volume_ext
- 0x00FE: glk_schannel_pause
- 0x00FF: glk_schannel_unpause
- 0x0100: glk_set_hyperlink
- 0x0101: glk_set_hyperlink_stream
- 0x0102: glk_request_hyperlink_event
- 0x0103: glk_cancel_hyperlink_event
- 0x0120: glk_buffer_to_lower_case_uni
- 0x0121: glk_buffer_to_upper_case_uni
- 0x0122: glk_buffer_to_title_case_uni

- 0x0123: glk_buffer_canon_decompose_uni
- 0x0124: glk_buffer_canon_normalize_uni
- 0x0128: glk_put_char_uni
- 0x0129: glk_put_string_uni
- 0x012A: glk_put_buffer_uni
- 0x012B: glk_put_char_stream_uni
- 0x012C: glk_put_string_stream_uni
- 0x012D: glk_put_buffer_stream_uni
- 0x0130: glk_get_char_stream_uni
- 0x0131: glk_get_buffer_stream_uni
- 0x0132: glk_get_line_stream_uni
- 0x0138: glk_stream_open_file_uni
- 0x0139: glk_stream_open_memory_uni
- 0x013A: glk_stream_open_resource_uni
- 0x0140: glk_request_char_event_uni
- 0x0141: glk_request_line_event_uni
- 0x0150: glk_set_echo_line_event
- 0x0151: glk_set_terminators_line_event
- 0x0160: glk_current_time
- 0x0161: glk_current_simple_time
- 0x0168: glk_time_to_date_utc
- 0x0169: glk_time_to_date_local
- 0x016A: glk_simple_time_to_date_utc
- 0x016B: glk_simple_time_to_date_local
- 0x016C: glk_date_to_time_utc
- 0x016D: glk_date_to_time_local
- 0x016E: glk_date_to_simple_time_utc
- 0x016F: glk_date_to_simple_time_local

Note that glk_main() does not have a selector, because it's provided by your program, not the library.

There is no way to use these selectors directly in the Glk API. *[An earlier version of Glk had gestalt selectors gestalt_FunctionNameToID and gestalt_FunctionIDToName, but these have been withdrawn.]* They are defined and used only by the dispatch layer.

Call selectors 0x1200 to 0x12FF are reserved for extension projects by Carlos Sanchez. The same is true of gestalt selector 0x1200. These are not documented here.

12.2. The Blorb Layer

The material described in this section is not part of the Glk API per se. It is an external layer which allows the library to load resources (images and sounds) from a file specified by your program. The Blorb file format is a standard IF resource archive.

The Glk spec does not require that resources be stored in a Blorb file. It says only that the library knows how to load them and use them, when you so request. However, Blorb is the recommended way to supply portable resources. Most Glk libraries will support Blorb, using the interface defined in this section.

The quick summary: resources are identified by type (image, sound, etc) and by an index number. *[But not by name. This is for historical reasons; Infocom's Z-machine architecture used this scheme.]*

For the complete Blorb specification and tools for Blorb file manipulation, see:

<http://eblong.com/zarf/blorb/>

12.2.1. How This Works

The Blorb layer is implemented in a C source file, gi_blorb.c, and its header, gi_blorb.h. This code is (mostly) platform-independent — it is identical in every library, just as the glk.h header file is identical in every library. Each library author who wants to support Blorb should download the gi_blorb.c and gi_blorb.h files from the Glk web site, and compile them unchanged into the library.

Most of the functions defined in gi_blorb.h are intended for the library. If you are writing a Glk program, you can ignore them all, except for giblorb_set_resource_map(); see section 12.2.2, "What the Program Does". If you are implementing a Glk library, you can use this API to find and load resource data.

12.2.2. What the Program Does

If you wish your program to load its resources from a Blorb file, you need to find and open that file in your startup code. (See section 11.1, "Startup Options".) Each platform will have appropriate functions available for finding startup data. Be sure to open the file in binary mode, not text mode. Once you have opened the file as a Glk stream, pass it to giblorb_set_resource_map().

```
   giblorb_err_t giblorb_set_resource_map(strid_t file);
```

This function tells the library that the file is indeed the Blorby source of all resource goodness. Whenever your program calls an image or sound function, such as glk_image_draw(), the library will search this file for the resource you request.

Do *not* close the stream after calling this function. The library is responsible for closing the stream at shutdown time.

If you do not call giblorb_set_resource_map() in your startup code, or if it fails, the library is left to its own devices for finding resources. Some libraries may try to load resources from individual files — PIC1, PIC2, PIC3, and so on. (See the Blorb specification for more on this approach.) Other libraries will not have any other loading mechanism at all; no resources will be available.

12.2.3. What the Library Does

Each library must implement giblorb_set_resource_map(), if it wishes to support Blorb at all. Generally, this function should create a Blorb map and stash it away somewhere. It may also want to stash the stream itself, so that the library can read data directly from it.

giblorb_set_resource_map() should return giblorb_err_None (0) if it succeeded, or the appropriate Blorb error code if not. See section 12.2.5, "Blorb Errors".

The library must also link in the gi_blorb.c file. Most of this should compile without difficulty on any platform. However, it does need to allocate memory. As supplied, gi_blorb.c calls the ANSI functions malloc(), realloc(), and free(). If this is not appropriate on your OS, feel free to change these calls. They are isolated at the end of the file.

12.2.4. What the Blorb Layer Does

These are the functions which are implemented in gi_blorb.c. They will be compiled into the library, but they are the same on every platform. In general, only the library needs to call these functions. The Glk program should allow the library to do all the resource handling.

```
   giblorb_err_t giblorb_create_map(strid_t file, giblorb_map_t **newmap);
```

This reads Blorb data out of a Glk stream. It does not load every resource at once; instead, it creates an map in memory which make it easy to find resources. A pointer to the map is stored in newmap. This is an opaque object; you pass it to the other Blorb-layer functions.

```
   giblorb_err_t giblorb_destroy_map(giblorb_map_t *map);
```

Deallocate the map and all associated memory. This does *not* close the original stream.

```
   giblorb_err_t giblorb_load_chunk_by_type(giblorb_map_t *map, glui32 method,
      giblorb_result_t *res, glui32 chunktype, glui32 count);
```

This loads a chunk of a given type. The count parameter distinguishes between chunks of the same type. If count is zero, the first chunk of that type is loaded, and so on.

To load a chunk of an IFF FORM type (such as AIFF), you should pass in the form type, rather than FORM. *[This introduces a slight ambiguity — you cannot distiguish between a FORM AIFF chunk and a non-FORM chunk of type AIFF. However, the latter is almost certainly a mistake.]*

The returned data is written into res, according to method:

```
   #define giblorb_method_DontLoad (0)
   #define giblorb_method_Memory (1)
   #define giblorb_method_FilePos (2)
   typedef struct giblorb_result_struct {
       glui32 chunknum;
       union {
           void *ptr;
           glui32 startpos;
       } data;
       glui32 length;
       glui32 chunktype;
   } giblorb_result_t;
```

The chunknum field is filled in with the number of the chunk. (This value can then be passed to giblorb_load_chunk_by_number() or giblorb_unload_chunk().) The length field is filled in with the length of the chunk in bytes. The chunktype field is the chunk's type, which of course will be the type you asked for.

If you specify giblorb_method_DontLoad, no data is actually loaded in. You can use this if you are only interested in whether a chunk exists, or in the chunknum and length parameters.

If you specify giblorb_method_FilePos, data.startpos is filled in with the file position of the chunk data. You can use glk_stream_set_position() to read the data from the stream.

If you specify giblorb_method_Memory, data.ptr is filled with a pointer to allocated memory containing the chunk data. This memory is owned by the map, not you. If you load the chunk more than once with giblorb_method_Memory, the Blorb layer is smart enough to keep just one copy in memory. You should *not* deallocate this memory yourself; call giblorb_unload_chunk()

instead.

```
   giblorb_err_t giblorb_load_chunk_by_number(giblorb_map_t *map, glui32
      method, giblorb_result_t *res, glui32 chunknum);
```

This is similar to giblorb_load_chunk_by_type(), but it loads a chunk with a given chunk number. The type of the chunk can be found in the chunktype field of giblorb_result_t. You can get the chunk number from the chunknum field, after calling one of the other load functions.

```
   giblorb_err_t giblorb_unload_chunk(giblorb_map_t *map, glui32 chunknum);
```

This frees the chunk data allocated by giblorb_method_Memory. If the given chunk has never been loaded into memory, this has no effect.

```
   giblorb_err_t giblorb_load_resource(giblorb_map_t *map, glui32 method,
      giblorb_result_t *res, glui32 usage, glui32 resnum);
```

This loads a resource, given its usage and resource number. Currently, the three usage values are giblorb_ID_Pict (images), giblorb_ID_Snd (sounds), and giblorb_ID_Exec (executable program). See the Blorb specification for more information about the types of data that can be stored for these usages.

Note that a resource number is not the same as a chunk number. The resource number is the sound or image number specified by a Glk program. Chunk number is arbitrary, since chunks in a Blorb file can be in any order. To find the chunk number of a given resource, call giblorb_load_resource() and look in res.chunknum.

```
   giblorb_err_t giblorb_count_resources(giblorb_map_t *map, glui32 usage,
      glui32 *num, glui32 *min, glui32 *max);
```

This counts the number of chunks with a given usage (image, sound, or executable.) The total number of chunks of that usage is stored in num. The lowest and highest resource number of that usage are stored in min and max. You can leave any of the three pointers NULL if you don’t care about that information.

12.2.5. Blorb Errors

All Blorb layer functions, including giblorb_set_resource_map(), return the following error codes.

- giblorb_err_None, or zero: No error.
- giblorb_err_CompileTime: Something is compiled wrong in the Blorb layer.
- giblorb_err_Alloc: Memory could not be allocated.
- giblorb_err_Read: Data could not be read from the file.
- giblorb_err_NotAMap: The map parameter is invalid.
- giblorb_err_Format: The Blorb file is corrupted or invalid.
- giblorb_err_NotFound: The requested data could not be found.