The Ghostscript Interpreter Application Programming Interface (API)

1 Table of contents

For other information, see the Ghostscript overview.

WARNING: The API described in this document is subject to changes in future releases, possibly ones that are not backward compatible with what is described here.

---

2 What is the Ghostscript Interpreter API?

The Ghostscript interpreter can be built as a dynamic link library (DLL) on Microsoft Windows, as a shared object on the Linux, Unix and MacOS X platforms. With some changes, it could be built as a static library. This document describes the Application Programming Interface (API) for the Ghostscript interpreter library. This should not be confused with the Ghostscript library which provides a graphics library but not the interpreter.

This supercedes the old DLL interface.

To provide the interface described in the usage documentation, a smaller independent executable loads the DLL/shared object. This executable must provide all the interaction with the windowing system, including image windows and, if necessary, a text window.

The Ghostscript interpreter library's name and characteristics differ for each platform:

- The Win32 DLL `gsdll32.dll` can be used by multiple programs simultaneously, but only once within each process.

- The OS/2 DLL `gsdll2.dll` has MULTIPLE NONSHARED data segments and can be called by multiple programs simultaneously.
- The Linux shared object `libgs.so` can be used by multiple programs simultaneously.

The source for the executable is in `dw`*.* (Windows), `dp`*.* (OS/2) and `dx`*.* (Linux/Unix). See these source files for examples of how to use the DLL.

The source file `dxmainc.c` can also serve as an example of how to use the shared library component on MacOS X, providing the same command-line tool it does on any linux, bsd or similar operating system.

At this stage, Ghostscript does not support multiple instances of the interpreter within a single process.

---

3 Exported functions

The functions exported by the DLL/shared object are described in the header file `iapi.h` and are summarised below. Omitted from the summary are the calling convention (e.g. __stdcall), details of return values and error handling.

- `int gsapi_revision (gsapi_revision_t *pr, int len);`
- `int gsapi_new_instance (void **pinstance, void *caller_handle);`
- `void gsapi_delete_instance (void *instance);`
- `int gsapi_set_stdio (void *instance, int(*stdin_fn)(void *caller_handle, char *buf, int len), int(*stdout_fn)(void *caller_handle, const char *str, int len), int(*stderr_fn)(void *caller_handle, const char *str, int len));`
- `int gsapi_set_poll (void *instance, int(*poll_fn)(void *caller_handle));`
- `int gsapi_set_display_callback (void *instance, display_callback *callback);`
- `int gsapi_set_arg_encoding (void *instance, int encoding);`
- `int gsapi_init_with_args (void *instance, int argc, char **argv);`
- `int gsapi_run_string_begin (void *instance, int user_errors, int *pexit_code);`
- `int gsapi_run_string_continue (void *instance, const char *str, unsigned int length, int user_errors, int *pexit_code);`
- `int gsapi_run_string_end (void *instance, int user_errors, int *pexit_code);`
- `int gsapi_run_string_with_length (void *instance, const char *str, unsigned int length, int user_errors, int *pexit_code);`
- `int gsapi_run_string (void *instance, const char *str, int user_errors, int *pexit_code);`
- `int gsapi_run_file (void *instance, const char *file_name, int user_errors, int *pexit_code);`
- `int gsapi_exit (void *instance);`
- `int gsapi_set_visual_tracer (gstruct vd_trace_interface_s *I);`

3.1 `gsapi_revision()`

This function returns the revision numbers and strings of the Ghostscript interpreter library; you should call it before any other interpreter library functions to make sure that the correct version of the Ghostscript interpreter has been loaded.

```
typedef struct gsapi_revision_s {
    const char *product;
    const char *copyright;
    long revision;
    long revisiondate;
} gsapi_revision_t;
gsapi_revision_t r;

if (gsapi_revision(&r, sizeof(r)) == 0) {
    if (r.revision < 650)
```

```
        printf("Need at least Ghostscript 6.50");
}
else {
    printf("revision structure size is incorrect");
}
```

3.2 `gsapi_new_instance()`

Create a new instance of Ghostscript. This instance is passed to most other gsapi functions. The caller_handle will be provided to callback functions. **Unless Ghostscript has been compiled with the GS_THREADSAFE define, only one instance at a time is supported.**

Historically, Ghostscript has only supported a single instance; any attempt to create more than one at a time would result in gsapi_new_instance returning an error. Experimental work has been done to lift this restriction; if Ghostscript is compiled with the GS_THREADSAFE define then multiple concurrent instances are permitted.

While the core Ghostscript devices are believed to be thread safe now, certain devices are known not to be (particularly the contrib devices). The makefiles currently make no attempt to exclude these from builds. If you enable GS_THREADSAFE then you should check to ensure that you do not rely on such devices (check for global variable use).

3.3 `gsapi_delete_instance()`

Destroy an instance of Ghostscript. Before you call this, Ghostscript must have finished. If Ghostscript has been initialised, you must call `gsapi_exit` before `gsapi_delete_instance`.

3.4 `gsapi_set_stdio()`

Set the callback functions for stdio The stdin callback function should return the number of characters read, 0 for EOF, or -1 for error. The stdout and stderr callback functions should return the number of characters written.

3.5 `gsapi_set_poll()`

Set the callback function for polling. This function will only be called if the Ghostscript interpreter was compiled with `CHECK_INTERRUPTS` as described in `gpcheck.h`.

The polling function should return zero if all is well, and return negative if it wants ghostscript to abort. This is often used for checking for a user cancel. This can also be used for handling window events or cooperative multitasking.

The polling function is called very frequently during interpretation and rendering so it must be fast. If the function is slow, then using a counter to return 0 immediately some number of times can be used to reduce the performance impact.

3.6 `gsapi_set_display_callback()`

Set the callback structure for the display device. If the display device is used, this must be called after `gsapi_new_instance()` and before `gsapi_init_with_args()`. See `gdevdsp.h` for more details.

3.7 `gsapi_set_arg_encoding()`

Set the encoding used for the interpretation of all subsequent args supplied via the gsapi interface on this instance. By default we expect args to be in encoding 0 (the 'local' encoding for this OS). On Windows this means "the currently selected codepage". On Linux this typically means utf8. This means that omitting to call this function will leave Ghostscript running exactly as it always has. Please note that use of the 'local' encoding is now deprecated and should be avoided in new code. This must be called after `gsapi_new_instance()` and before `gsapi_init_with_args()`.

3.8 `gsapi_init_with_args()`

Initialise the interpreter. This calls `gs_main_init_with_args()` in `imainarg.c`. See below for return codes. The arguments are the same as the "C" main function: argv[0] is ignored and the user supplied arguments are argv[1] to argv[argc-1].

3.9 `gsapi_run_*()`

The `gsapi_run_*` functions are like `gs_main_run_*` except that the error_object is omitted. If these functions return <= -100, either quit or a fatal error has occured. You must call `gsapi_exit()` next. The only exception is `gsapi_run_string_continue()` which will return `gs_error_NeedInput` if all is well. See below for return codes.

The address passed in `pexit_code` will be used to return the exit code for the interpreter in case of a quit or fatal error. The `user_errors` argument is normally set to zero to indicate that errors should be handled through the normal mechanisms within the interpreted code. If set to a negative value, the functions will return an error code directly to the caller, bypassing the interpreted language. The interpreted language's error handler is bypassed, regardless of `user_errors` parameter, for the `gs_error_interrupt` generated when the polling callback returns a negative value. A positive `user_errors` is treated the same as zero.

There is a 64 KB length limit on any buffer submitted to a `gsapi_run_*` function for processing. If you have more than 65535 bytes of input then you must split it into smaller pieces and submit each in a separate `gsapi_run_string_continue()` call.

3.10 `gsapi_exit()`

Exit the interpreter. This must be called on shutdown if `gsapi_init_with_args()` has been called, and just before `gsapi_delete_instance()`.

3.11 `gsapi_set_visual_tracer()`

Pass visual tracer interface block to Ghostscript. See `Lib.htm` for reference about the interface block structure. This function is useful for debug clients only. Release clients must not call it.

3.12 Return codes

The `gsapi_init_with_args`, `gsapi_run_*` and `gsapi_exit` functions return an integer code.

Return codes from `gsapi_*()`

Code	Status
0	No errors
gs_error_Quit	"`quit`" has been executed. This is not an error.

	`gsapi_exit()` must be called next.
gs_error_interrupt	The polling callback function returned a negative value, requesting Ghostscript to abort.
gs_error_NeedInput	More input is needed by `gsapi_run_string_continue()`. This is not an error.
gs_error_Info	"`gs -h`" has been executed. This is not an error. `gsapi_exit()` must be called next.
< 0	Error
<= gs_error_Fatal	Fatal error. `gsapi_exit()` must be called next.

The `gsapi_run_*()` functions do not flush stdio. If you want to see output from Ghostscript you must do this explicitly as shown in the example below.

When executing a string with `gsapi_run_string_*()`, `currentfile` is the input from the string. Reading from `%stdin` uses the stdin callback.

4 Example Usage

To try out the following examples in a development environment like Microsoft's developer tools or Metrowerks Codewarrior, create a new project, save the example source code as a `.c` file and add it, along with the Ghostscript dll or shared library. You will also need to make sure the Ghostscript headers are available, either by adding their location (the `src` directory in the Ghostscript source distribution) to the project's search path, or by copying ierrors.h and iapi.h into the same directory as the example source.

4.1 Example 1

```
/* Example of using GS DLL as a ps2pdf converter.  */

#if defined(_WIN32) && !defined(_Windows)
# define _Windows
#endif
#ifdef _Windows
/* add this source to a project with gsdll32.dll, or compile it directly with:
 *   cl -D_Windows -Isrc -Febin\ps2pdf.exe ps2pdf.c bin\gsdll32.lib
 */
# include <windows.h>
# define GSDLLEXPORT __declspec(dllimport)
#endif

#include "ierrors.h"
#include "iapi.h"

void *minst;

int main(int argc, char *argv[])
{
    int code, code1;
    const char * gsargv[10];
    int gsargc;
    gsargv[0] = "ps2pdf";       /* actual value doesn't matter */
    gsargv[1] = "-dNOPAUSE";
    gsargv[2] = "-dBATCH";
    gsargv[3] = "-dSAFER";
    gsargv[4] = "-sDEVICE=pdfwrite";
    gsargv[5] = "-sOutputFile=out.pdf";
    gsargv[6] = "-c";
    gsargv[7] = ".setpdfwrite";
    gsargv[8] = "-f";
    gsargv[9] = "input.ps";
    gsargc=10;

    code = gsapi_new_instance(&minst, NULL);
    if (code < 0)
        return 1;
    code = gsapi_set_arg_encoding(minst, GS_ARG_ENCODING_UTF8);
    if (code == 0)
        code = gsapi_init_with_args(minst, gsargc, gsargv);
    code1 = gsapi_exit(minst);
    if ((code == 0) || (code == gs_error_Quit))
        code = code1;

    gsapi_delete_instance(minst);

    if ((code == 0) || (code == gs_error_Quit))
        return 0;
    return 1;
}
```

4.2 Example 2

```
/* Similar to command line gs */

#if defined(_WIN32) && !defined(_Windows)
# define _Windows
#endif
#ifdef _Windows
/* Compile directly with:
 *   cl -D_Windows -Isrc -Febin\gstest.exe gstest.c bin\gsdll32.lib
 */
# include <windows.h>
# define GSDLLEXPORT __declspec(dllimport)
#endif
#include <stdio.h>
#include "ierrors.h"
#include "iapi.h"

/* stdio functions */
static int GSDLLCALL
gsdll_stdin(void *instance, char *buf, int len)
{
    int ch;
    int count = 0;
    while (count < len) {
        ch = fgetc(stdin);
        if (ch == EOF)
            return 0;
        *buf++ = ch;
        count++;
        if (ch == '\n')
            break;
    }
    return count;
}

static int GSDLLCALL
gsdll_stdout(void *instance, const char *str, int len)
{
    fwrite(str, 1, len, stdout);
    fflush(stdout);
    return len;
}

static int GSDLLCALL
gsdll_stderr(void *instance, const char *str, int len)
{
    fwrite(str, 1, len, stderr);
    fflush(stderr);
    return len;
}
```

```
void *minst;
const char start_string[] = "systemdict /start get exec\n";

int main(int argc, char *argv[])
{
    int code, code1;
    int exit_code;

    code = gsapi_new_instance(&minst, NULL);
    if (code < 0)
        return 1;
    gsapi_set_stdio(minst, gsdll_stdin, gsdll_stdout, gsdll_stderr);
    code = gsapi_set_arg_encoding(minst, GS_ARG_ENCODING_UTF8);
    if (code == 0)
        code = gsapi_init_with_args(minst, argc, argv);
    if (code == 0)
        code = gsapi_run_string(minst, start_string, 0, &exit_code);
    code1 = gsapi_exit(minst);
    if ((code == 0) || (code == gs_error_Quit))
        code = code1;

    gsapi_delete_instance(minst);

    if ((code == 0) || (code == gs_error_Quit))
        return 0;
    return 1;
}
```

4.3 Example 3

Replace main() in either of the above with the following code, showing how you can feed Ghostscript piecemeal:

```
const char *command = "1 2 add == flush\n";

int main(int argc, char *argv[])
{
    int code, code1;
    int exit_code;

    code = gsapi_new_instance(&minst, NULL);
    if (code < 0)
        return 1;
    code = gsapi_set_arg_encoding(minst, GS_ARG_ENCODING_UTF8);
    if (code == 0)
        code = gsapi_init_with_args(minst, argc, argv);

    if (code == 0) {
        gsapi_run_string_begin(minst, 0, &exit_code);
        gsapi_run_string_continue(minst, command, strlen(command), 0, &exit_code);
        gsapi_run_string_continue(minst, "qu", 2, 0, &exit_code);
        gsapi_run_string_continue(minst, "it", 2, 0, &exit_code);
        gsapi_run_string_end(minst, 0, &exit_code);
    }

    code1 = gsapi_exit(minst);
    if ((code == 0) || (code == gs_error_Quit))
        code = code1;

    gsapi_delete_instance(minst);

    if ((code == 0) || (code == gs_error_Quit))
        return 0;
    return 1;
}
```

4.4 Example 4

When feeding Ghostscript piecemeal buffers, one can use the normal operators to configure things and invoke library routines. For example, to parse a PDF file one could say:

```
    code = gsapi_run_string(minst, "(example.pdf) .runlibfile", 0, &exit_code);
```

and Ghostscript would open and process the file named "example.pdf" as if it had been passed as an argument to `gsapi_init_with_args()`.

5 Multiple threads

The Ghostscript library should have been compiled with a thread safe run time library. Synchronisation of threads is entirely up to the caller. The exported `gsapi_*()` functions must be called from one thread only.

6 Standard input and output

When using the Ghostscript interpreter library interface, you have a choice of two standard input/output methods.

- If you do nothing, the "C" stdio will be used.
- If you use `gsapi_set_stdio()`, all stdio will be redirected to the callback functions you provide. This would be used in a graphical user interface environment where stdio is not available, or where you wish to process Ghostscript input or output.

The callback functions are described in `iapi.h`.

7 Display device

The `display` device is available for use with the Ghostscript interpreter library. This is described in the file `gdevdsp.h`. This device provides you with access to the raster output of Ghostscript. It is your responsibility to copy this raster to a display window or printer.

To use this device, you must provide a callback structure with addresses of a number of callback functions. The address of the callback structure is provided using `gsapi_set_display_callback()`. This must be called after `gsapi_new_instance()` and before `gsapi_init_with_args()`.

The callbacks are for device open, close, resize, sync, page, memory allocation and updating. Each callback function contains a handle can be set using

-sDisplayHandle=1234

Where "1234" is a string. The API was changed to use a string rather than an integer/long value when support for 64 bit systems arrived. A display "handle" is often a pointer, and since these command line options have to survive being processed by Postscript machinery, and Postscript only permits 32 bit number values, a different representation was required. Hence changing the value to a string, so that 64 bit values can be supported. The string formats allowed are:

`1234` - implicit base 10

`10#1234` - explicit base 10

`16#04d2` - explicit base 16

The "number string" is parsed by the display device to retrieve the number value, and is then assigned to the void pointer parameter "pHandle" in the display device structure. Thus, for a trivial example, passing `-sDisplayHandle=0` will result in the first parameter passed to your display device callbacks being: `(void *)0`.

The previous API, using a number value:

    -dDisplayHandle=1234

is still supported on 32 bit systems, but will cause a "typecheck" error on 64 bit systems, and is considered deprecated. It should not be used in new code.

The device raster format can be configured using

    -dDisplayFormat=NNNN

Options include

- native, gray, RGB, CMYK or separation color spaces.
- alpha byte (ignored).
- 1 to 16 bits/component.
- bigendian (RGB) or littleendian (BGR) order.
- top first or bottom first raster.
- 16 bits/pixel with 555 or 565 bitfields.

The format values are described in **`gdevdsp.h`**. The format is flexible enough to support common Windows, OS/2, Linux and Mac raster formats. To select the display device with a Windows 24-bit RGB raster:

```
    char **nargv;
    char arg1[64];
    char arg2[64];
    char arg3[64];
    code = gsapi_new_instance(&minst, NULL);
    gsapi_set_stdio(minst, gsdll_stdin, gsdll_stdout, gsdll_stderr);
    code = gsapi_set_display_callback(minst, &display_callback);
    sprintf(arg1, "-sDEVICE=display");
    sprintf(arg2, "-dDisplayHandle=%d", 0);
    sprintf(arg3, "-dDisplayFormat=%d",
        DISPLAY_COLORS_RGB | DISPLAY_ALPHA_NONE | DISPLAY_DEPTH_8 |
        DISPLAY_LITTLEENDIAN | DISPLAY_BOTTOMFIRST);
    nargv = (char **)malloc((argc + 4) * sizeof(char *));
    nargv[0] = argv[0];
    nargv[1] = arg1;
    nargv[2] = arg2;
    nargv[3] = arg3;
    memcpy(nargv + 4, argv + 1, argc * sizeof(char *));
    argc += 3;
    code = gsapi_init_with_args(minst, argc, nargv);
```

The display device provides you with the address and size of the raster using the **`display_size()`** callback. You are then responsible for displaying this raster. Some examples are in **`dwmain.c`** (Windows), **`dpmain.c`** (OS/2) and **`dxmain.c`** (X11/Linux), and **`dmmain.c`** (MacOS Classic or Carbon).

On some platforms, the calling convention for the display device callbacks in **`gdevdsp.h`** is not the same as the exported **`gsapi_*()`** functions in **`iapi.h`**.

Ghostscript version 9.19, 23 March 2016