



DrWimpC URL Launcher Library.



[Next](#)

Introduction.

This is a small library that provides a URL launcher facility for DrWimpC programs. It is built using the DrWimpC library. As a test, DrWimpC Application Builder uses it to launch the DrWimpC web site page from the new 'Web' button in its Program Info box. Four versions of the URL launcher library are included in the zip file archive. Two are compiled using the ROOL C tools, one with debug switched on. The other two are compiled with GCC, one static and the other a shared library. Full source code is included containing the single interface function `drw_launch_url()` and its support functions that implement the ANT and Acorn URI launch protocols. In providing this library I have looked around the web for information on methods to implement a URL launcher program on Risc Os. Mostly I found old code examples for the ANT protocol. The most instructive being the DeskLib code in `url.c`. Inevitably, there is some similarity with that code and this implementation. And after all, it's a simple piece of code implementing a simple mechanism so there aren't many different approaches to consider. I decided that to get things working I'd keep to the guidelines of the examples I found. In any case I've provided the source code so you can remove any redundant functionality. (e.g. If all that's needed is the Acorn URI stuff). It's also a chance to test and illustrate the use of using user assigned event handlers - normally either a user function or template function is used to handle the application specific code for an event but you can also register and unregister an event handling function in DrWimpC just as you can with other event processing libraries. This is most useful for temporary situations like using NULL events when the need arises.

The General Approach.

Most of the code I found either didn't deal with long URLs (Those that don't fit into a wimp message), or made a distinction between the shorter URLs and the long ones which needed to be stored in memory that wouldn't be paged out (usually RMA) so that the receiver of the URL launch request can access them. I decided to always store the URL in a small heap in a dynamic area (DA) (This also demonstrates the use of a heap in DrWimpC library).

The procedure tries both the ANT and Acorn URL launch method and begins with the ANT one by setting up and broadcasting the ANT URL launch message (message `INET_SUITE_OPEN_URL`, &4AF80) as a user message recorded so that if no application receives nor processes the message then the message is returned to your application as a user message acknowledge. If that occurs then the Acorn URI launch is tried.. To catch the user message acknowledge event a handler routine is registered with the DrWimpC library prior to sending the message.

If an application processes the message then the URL has been successfully launched. There is no indication of this so an event handler for NULL events is registered with the DrWimpC library prior to sending the message. If the NULL event handler gets called then the URL has been successfully launched.

In both cases the event handlers are unregistered since they are only needed for the duration of the URL launch.

The Acorn URI launch method has a return message acknowledgment indicating success or failure. The message is message `URI_RETURN_RESULT`, &4E383). A handler is registered with the DrWimpC library to deal with receipt of the message prior to calling `swi_URI_dispatch` (&4E381) using the X form of the `swi` with flag `uri_DISPATCH_INFORM_CALLER` (&1) set in `r0` on input.

If there is an error in the `swi` call or it sets the return flag `uri_DISPATCH_NOT_CLAIMED` (&1) in the return result flags in `r0` then the URL wasn't launched. In that case there is a last chance to launch the URL by attempting to run a program that can handle its protocol (`http:`, etc). To do this the URL is examined to identify its protocol based on finding a colon in the URL string that delimits the protocol name. Then a system variable name is built from it of the form `Alias$URLOpen_<protocol name>`. If this exists it is used to launch a program to process the URL.

This is also done if the registered handler for message `URI_RETURN_RESULT` is called and the URL has not been claimed (the message flags will have the flag `uri_DISPATCH_NOT_CLAIMED` unset (a good double negative).

Communicating the Result to Your Application.

The process can succeed or fail at various points and it may be useful to inform your application at the time so it can take action. A natural way to do this is to call a function with parameters indicating success or failure, and this is the approach used here. There is a function pointer parameter in the call to `drw_launch_url()` allowing your application to specify a function to be called when the URL launch succeeds or fails. This pointer is stored with the URL to be launched so it can be a URL specific function.

The Library's Application Interface.

This is the single function `drw_launch_url()` defined as:

```
int drw_launch_url(char *url,
                   drw_url_fn urldonefn,
                   int maxheapsize);
```

`char *url` is the URL string.

`drw_url_fn` is a pointer to an application defined function to be called when the URL launch succeeds or fails.

Its typedef is `void (*drw_url_fn)(char *url, int done);` defined in `DRWURL:h.drwurth`

`char *url` is the URL string being launched.

`int done` is the result. TRUE for a successful launch, FALSE for failure.

`int maxheapsize` is the maximum size that you want the DA holding URL data to become. This will mainly depend on the maximum length of the URLs you launch and if for some reason you launch more than one in any `wimp_poll` of your application. (The command generated from the `Alias$URLOpen_` variable is also conveniently stored in the heap, so will need to be accounted for in the `maxheapsize`).

The function's return value is the return value from the call to `drw_broadcast_ant_url()` which is an `os_error` pointer, NULL for success, otherwise a valid pointer to an error block.. Note that if the memory allocation fails no error is generated but the user function specified in the `urldonefn` pointer is called, so you should use that to determine the success or not of the URL launch.

The Library's URL Data Structure.

The data structure simply consists of the `drw_url_fn` pointer from the `drw_launch_url` call and the URL string.

It is defined in `DRWURL:h.drwurth` -

```
struct {
    drw_url_fn urldonefn;
    char url[UNKNOWN];
} drw_url_data;
```

UNKNOWN is defined in `OSLib:h.types` as 1. The actual URL string is stored at the end of the `drw_url_data` structure to allow for variable length URL strings.

An `OS_Heap` heap is implemented in a dynamic area to store the data structure for each URL. They are freed when the URL launch succeeds or fails.

The Library's Static Data.

Three items are defined. One to store the pointer to the dynamic area for the heap of URL strings and one to point to the current URL command generated from an `Alias$URLOpen_` system variable (there will only ever be one of these at any time)., and one to track the size of command so it can be resized as necessary.

The definitions are -

```
static byte *drwurldarea = NULL;
static char *drwurcmd = NULL;
static int drwlurcmdsize = 0;
```

Source Code Program Notes.

Function `drw_launch_url`.

Its main task is to store the URL and the associated pointer to an application defined success or failure function and then call `drw_broadcast_ant_url()` to start the URL launch process. On first call it creates a new heap by calling `drw_wimp_create_os_heap()` which creates a dynamic area using `drw_wimp_createdynamic()` and then calls `drw_new_os_heap()` for the dynamic area to initialise the heap. The dynamic area is given a name of `<App name>_urls_buffer`. Then, `drw_os_heap_alloc()` allocates space from the heap for a `drw_url_data` structure for the URL with the space required rounded up to a whole number of words. Finally, `drw_broadcast_ant_url()` is called.

Function `drw_broadcast_ant_url`.

The ANT URL launch method is attempted by setting up the message_INET_SUITE_OPEN_URL. The C structure of the message data is a union of data structures for short URLs (the direct structure) and the indirect structure for a long URL. For the former the URL is copied into the message data and the latter a pointer to the URL data is used with all other data elements initialised to default values. Note the difference in the message sizes. The first is the size of the message header plus the length of the URL string rounded up to a whole number of words, the second is the size of the full indirect message structure.

Prior to sending the message two functions are registered with the DrWimpC library, one to catch a returned unprocessed message - drw_catch_rtnd_ant_url_msg() - and the second - drw_url_catch_nulls() - to process the first NULL event received by the application after sending the message, which would indicate a successful URL launch. Note that for NULL events to be received by your application drw_app_start_null_events() should be called and to stop receiving them the corresponding function drw_app_stop_null_events() should be called. These functions and the event handler registration functions are described in the DrWimpC Functional API StrongHelp manual.

As usual, if an error occurs the application is informed through its specified success or failure function, and the data freed and event handlers deregistered and NULL events cancelled.

Function drw_url_catch_nulls

This function will be called when the application receives a NULL event. At this point the URL launch has succeeded so all that needs to be done is inform the application and clear up by releasing the registered functions and freeing the URL data. This function will only be called once at most. If it doesn't get called the URL launch has failed for some reason at which point the function will be unregistered and so will not be called..

Function drw_catch_rtnd_ant_url_msg.

message_INET_SUITE_OPEN_URL is broadcast as a User Message Recorded (Event type 18) so that if no application deals with it, it is returned to your application as an event type 19 (User Message Acknowledge). (This function is no longer required and is unregistered by calling drw_release_ant_fn, along with drw_url_catch_nulls).

So now the Acorn URI launch method is tried. Since we need to determine if the Acorn URI launch fails - usually indicating that no application has been seen by the Risc Os filer that will handle the URL, the function drw_catch_acorn_open_url_msg() is registered to receive the message message_URI_RETURN_RESULT, &4E383), and the URI_Dispatch swi is called with ithe input flag uri_DISPATCH_INFORM_CALLER (&1) set.

If the swi returns an error or the output flag uri_DISPATCH_NOT_CLAIMED (&1) is set then the URI launch has failed for some reason or was not processed by an application. So a final attempt to launch the URL is made by calling drw_start_url_loader() to try and directly run an application that will handle the URL.

Note the swi call conditional compile depending on whether _DRW_USE_SWI_FN_ is defined in the make file. If it is not defined then the OSlib function that calls swi URI_dispatch is used, otherwise the generic _swix() function is used to call the X form of the swi. This is to accommodate the shared library version of the drwurl library.

Function drw_catch_acorn_open_url_msg.

If this function is called then the URI_dispatch swi was successfully processed and the Acorn URI module is informing the application of the outcome. If the message flag uri_DISPATCH_NOT_CLAIMED (&1) is not set then the URL has been successfully launched and the application can be informed. The URL data can be freed.

Once called this function is no longer required and can be unregistered.

If the uri_DISPATCH_NOT_CLAIMED (&1) flag was set then no application was found to launch the URL. So try to run an application directly by calling drw_start_url_loader().

Function drw_start_url_loader.

As a final attempt to launch the URL this function identifies the URL protocol (delimited by a ':' e.g. http, https, etc).

If found a system variable name Alias\$URLOpen_<protocol type> is constructed. If the system variable exists, it is used to start an application. This is what the following code sequence from the function is doing:

```
strcpy(drwurlcmd, "Alias$URLOpen_");
strncat(drwurlcmd, url->url, cmdtypesize);
*(drwurlcmd + 14 + cmdtypesize) = '\0';
```

```
if ((newurlcmdsize = drw_wimp_testsysvariable(drwurlcmd)))
{
    strcat(drwurlcmd, " ");
```

```
strcat(drwurlcmd, url->url);
```

```
if (!drw_wimp_start_task((drwurlcmd + 6), NULL))
```

```
done = TRUE;
```

```
}
```

The first three lines append the URL's protocol type to the string Alias\$URLOpen_ (which is 14 characters long - which explains the '14' of the third statement). The variable name is passed to drw_wimp_testsysvariable() which returns a non zero length if the system variable exists, and has a value.

Then, the command to run is constructed as "Alias\$URLOpen_<protocol name>" followed by a space and then the URL itself. drw_wimp_start_task() runs the command. The '+ 6' ensures the not needed 'Alias\$' at the beginning of the command string is not included in the command string that is run. So the command string that is run has the form URLOpen_<protocol name> <URL string>. The operating system will use the value of the system variable Alias\$URLOpen_<protocol name> with the URL string as parameter as the command to run.

Functions drw_url_release_acorn_fn and drw_url_release_ant_fn.

These two functions are worth mentioning as they illustrate the use of drw_release_user_action(), drw_release_user_msg_action() and drw_release_user_msg_ack() to unregister application event handler functions.

Functions drw_init_url_data and drw_free_url_data.

These two functions complete the library and do what their names say..

Compiling the Library.

Four versions of the library are catered for. Two for the ROOL C tools, one with debug information included. And two for the GCC compiler, one statically compiled, the other compiled as a shared library. The four files !Make32, !Make32D, !Make32Ga and !Make32Gs should be run to compile each type of the library. They run the acorn make files DRWURLlib32, DRWURLLIB32D, DRWURLLIB32Ga and DRWURLLIB32Gs respectively. Running the DRWURLLIB* files directly will cause an error message of the form 'Make file format Incorrect - removing project'.

The ROOL C tools make files compile into the o and od directories and store the compiled libraries into those directories. The GCC make files compile into the oga and os directories and store the libraries in the !drwurl directory.

Two symlink files for the shared library are provided in !drwurl - libdrwURLlib32Gs/so and libdrwURLlib32Gs/so/1 for pointing to the compile time and run time shared library.

Next

The !Boot file sets the DRWURL\$Dir and DRWURL\$Path system variables so that header files in the sub directory !drwurl.DRWURL can be referenced in the source file by the usual method of

DRWURL/<header file name>.h, for example #include "DRWURL/drwurth.h".



DrWimpC URL Launcher Library.



[Previous](#) **Library Location.**

Since the shared version of the library is used by DrWimpC Application Builder, the application directory containing the shared version (!drwurl) must have been seen by the filer before DrWimpC Application Builder is run. So, !drwurl is placed inside the !DRWDEF.Repository directory so that !drwurl.!Boot is run on machine boot and the system variable DRWURL\$Path will be set.

A side effect of placing !drurl inside !DRWDEF.Repository is that it will have an entry in the DrWimpC Application Builder 'Objects' list, but having no objects. You can use this entry to open the !drwurl directory, or the !drwUo application supplied in the DrWimpC zip archive to open !drwurl (after !drwurl has been seen by the filer).

When compiling a new shared version of the library keep in mind that for DrWimpC Application Builder to us it you will need to either use the current !drwurl in !DRWDEF move it out of the way and make sure the new version has been seen by the filer before running DrWimpC Application Builder. If you use a different location, !drwurl will no longer appear in the DrWimpC Application Builder 'Objects' list.

Library Details.

System Variables:

DRWURL\$Dir Set to the Location of !drwurl.
DRWURL\$Path Set to <DRWURL\$Dir> plus a '.'

Default Location !DRWDEF.Repository

Shared Version libdrwURLlib32Gs/so/1/0/0
(Symlinks) libdrwURLlib32Gs/so and libdrwURLlib32Gs/so/1