

---

[Pages](#) / [Clarion Content Home](#)

## The ClarionMag Library (CML)

Added by David Harms, last edited by David Harms on May 04, 2014

With the return to the ClarionMag brand for this site it became necessary to rebrand the DevRoadmaps Clarion Library (DCL) as the ClarionMag Library (CML) and give it a new repository in GitHub.

[Read about the changes here.](#)

### **DCL to CML changes under way**

We've copied the DCL docs to this new location and are still in the process of scrubbing the pages and fixing all the text and links. If you notice anything amiss feel free to please post a comment at the bottom of the page.

Thanks!

The ClarionMag Library (CML) is a collection of classes gleaned from the original Clarion Magazine site as well as new source from this site. It includes a variety of utility classes as well as [ClarionTest \(CML\)](#), a unit testing framework created specifically for Clarion.

### **How to get the repository**

See [The ClarionMag Library \(CML\) on GitHub](#).

### **How to use the library**

See [How to use the ClarionMag Library \(CML\)](#). You may also want to read about the [CML directory tree structure](#). If you're new to object-oriented programming, check out [Object oriented programming in Clarion](#).

### **Documentation**

We're working on the class library docs, [which you can find here](#).

- [Rebranding the DCL to the CML, and other changes](#)
- [The ClarionMag Library \(CML\) on GitHub](#)
- [How to use the ClarionMag Library \(CML\)](#)
- [Look Ma! No project defines!](#)
- [CML directory tree structure](#)
- [ClarionTest](#)
- [CML docs](#)
- [DCL to CML changes under way](#)

[Like](#) Be the first to like this

No labels

[Pages](#) / [Clarion Content Home](#) / [The ClarionMag Library \(CML\)](#)

## CML directory tree structure

Added by David Harms, last edited by David Harms on May 04, 2014

### DCL to CML changes under way

We've copied the DCL docs to this new location and are still in the process of scrubbing the pages and fixing all the text and links. If you notice anything amiss feel free to please post a comment at the bottom of the page. Thanks!

The DCL directory tree structure looks like this (with the possibility of later additions):



This directory structure is designed to serve the purposes of the DCL, not necessarily to mimic the Clarion approach in every detail.

### .git

The .git directory (which you may not see as it's hidden) contains your local Git DCL repository.

### bin

The bin directory contains DevRoadmapsClarion.DLL, which is automatically copied from the build directory.

You won't see any "win" subdirectories, e.g. under bin, lib, libsrc, and template. I thought about putting them in but they seemed like unnecessary clutter since these directories live outside the Clarion directories anyway.

### build

The build directory contains the project used to create DevRoadmapsClarion.DLL and DevRoadmapsClarion.LIB.

### clariontest

Unit testing is an important part of almost all my class development, so yes, the DCL includes all the source code for the ClarionTest utility. Until recently ClarionTest was an app, but to make it easier to track changes (and avoid unexpected dependencies on templates that may or may not be regis<sup>t</sup>ered) I've moved it to source code only. There's still a lot of cleaning up to do.

### lib

The bin directory contains DevRoadmapsClarion.LIB, which is automatically copied from the build's release or debug directory, whichever has the more recent copy of the file.

## **libsrc**

The libsrc directory contains all of the class and associated source files.

## **template**

The template directory is home to the templates needed to create ClarionTest DLLs.

## **UnitTests**

The UnitTests directory contains ClarionTest.exe and its supporting DLLs. Individual unit test apps are in their respective subdirectories.

[Like](#) Be the first to like this

No labels

[Pages](#) / [Clarion Content Home](#) / [The ClarionMag Library \(CML\)](#)

## How to use the ClarionMag Library (CML)

Added by David Harms, last edited by David Harms on May 13, 2014

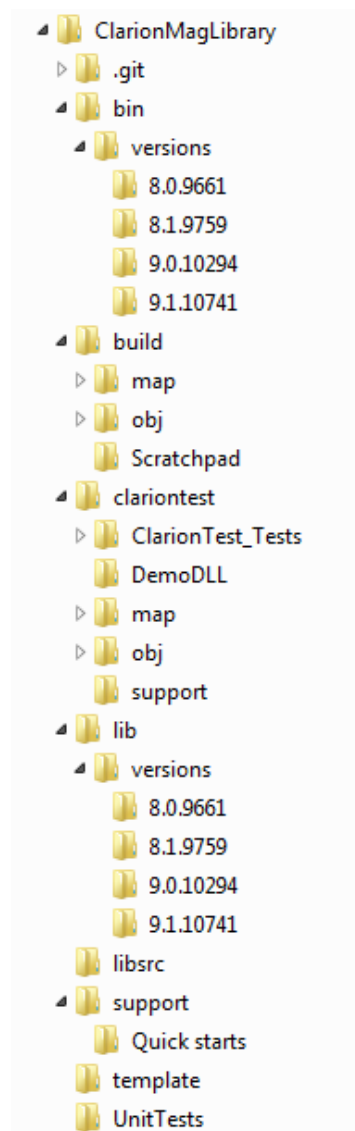
### DCL to CML changes under way

We've copied the DCL docs to this new location and are still in the process of scrubbing the pages and fixing all the text and links. If you notice anything amiss feel free to please post a comment at the bottom of the page. Thanks!

### Downloading the CML

If you haven't already done so, either [download the CML](#) or (preferably) [clone the repository using GitHub for Windows](#).

You'll now have a directory tree that looks something like this:



You won't see the .git directory unless you have Windows Explorer configured to show hidden files/folders.

## The RED file: To modify or not to modify

In order to use the CML files you need to find some way of getting those files into Clarion's field of view. You have two choices:

1. Copy the files to the appropriate Clarion directories
2. Modify Clarion's redirection file to include the appropriate CML directories

I favor the latter approach because it means that Clarion is always using the latest CML files. The downside is I need to modify the redirection file.

There are four lines that have to be modified in the Clarionx0.red file (which you can find in Clarion's bin directory). Here is what the changes look like in my RED.

```
[Common]
*.chm = %BIN%;%ROOT%\Accessory\bin
*.tp? = %ROOT%\template\win; F:\Clarion\ClarionMagLibrary\template
*.trf = %ROOT%\template\win
*.txs = %ROOT%\template\win
*.stt = %ROOT%\template\win
*. * = .; %ROOT%\libsrc\win; %ROOT%\images; %ROOT%\template\win;
F:\Clarion\ClarionMagLibrary\libsrc
*.lib = %ROOT%\lib; F:\Clarion\ClarionMagLibrary\lib
*.obj = %ROOT%\lib
*.res = %ROOT%\lib
*.hlp = %BIN%;%ROOT%\Accessory\bin
*.dll = %BIN%;%ROOT%\Accessory\bin; F:\Clarion\ClarionMagLibrary\bin
*.exe = %BIN%;%ROOT%\Accessory\bin; F:\Clarion\ClarionMagLibrary\bin
*.tp? = %ROOT%\Accessory\template\win
*.txs = %ROOT%\Accessory\template\win
*.stt = %ROOT%\Accessory\template\win
*.lib = %ROOT%\Accessory\lib
*.obj = %ROOT%\Accessory\lib
*.res = %ROOT%\Accessory\lib
*.dll = %ROOT%\Accessory\bin
*. * = %ROOT%\Accessory\images; %ROOT%\Accessory\resources; %ROOT%\Accessory\libsrc\win;
%ROOT%\Accessory\template\win
```

I know, I should write a utility to update the RED.

## Using the library

At present the CML is more of a hand coder's library, but if you're strictly an AppGen developer it's not that hard to take advantage of CML functionality. And yes, just like the RED utility, a comprehensive template to make it easier to use CML features is on my ToDo list.

There are two ways to use the classes; you can compile the source code, or you can use the precompiled DLL.

In either case, to make use of any of the classes you need to include the appropriate .INC file somewhere in your source, e.g.

```
Include('CML_System_String.inc'),once
```

Make sure you use the ,ONCE attribute.

By default the CML assumes you want to use the supplied LIB and DLL, so you'll need to add CMagLib.lib to your project. If you don't do this you'll get any number of unresolved external error messages.

### Which version of the DLL/LIB do I need?

The default compiled version of the library is for the latest Clarion release (or at least the latest one I happen to have installed on my machine). As of this writing, that means Clarion 9.1. If you're using an earlier version of Clarion have a look under the bin\versions and lib\versions directories for a suitable LIB and DLL. You may want to update your RED file to look in those directories.

### If you want to compile the classes

If you want to compile the classes you won't need the LIB and DLL, but you will need to set a compile pragma in your project properties:

```
_Compile_CML_Class_Source_=>1
```

Your builds will take a little longer this way, which isn't a big hassle especially if you have just one APP that uses CML. It's more of a problem if you're using CML with multiple apps. If there are any changes to CML code you need to compile each and every one of those apps. Unless you really want to compile the source I suggest using CML in its default configuration.

[Like](#) Be the first to like this

No labels

[Pages](#) / [Clarion Content Home](#) / [The ClarionMag Library \(CML\)](#)

## Look Ma! No project defines!

Added by David Harms, last edited by David Harms on May 04, 2014

### DCL to CML changes under way

We've copied the DCL docs to this new location and are still in the process of scrubbing the pages and fixing all the text and links. If you notice anything amiss feel free to please post a comment at the bottom of the page. Thanks!

The DCL is all about classes. And there are fundamentally two ways to use a class: you can compile the class into the application itself, or you can compile it into a library (usually a DLL) so the class can be used in a precompiled form.

The disadvantage of the first approach is that if you need to make a change to the class, say a bug fix to one of the methods, you need to recompile every app that uses that class. The advantage is it's harder to GPF your app using this technique.

The disadvantage of the second approach is that you have to add some extra information to the class definition, and you have to add some extra project defines to each and every APP (or hand coded project) that uses the class DLL, and if you get any of this wrong your app will almost surely GPF. The advantage is that if you make a change to the class that doesn't affect any of its existing exports, you don't need to recompile all the apps that use that class. You just drop in the new DLL.

### An ABC example

Here's a typical ABC class header:

```
StepClass      CLASS,MODULE('ABBROWSE.CLW'),TYPE|
               ,LINK('ABBROWSE.CLW',_ABCLinkMode_),DLL(_ABCDllMode_)
```

Note the `LINK` and `DLL` attributes. These are the little critters that cause so much grief. They're there because a DLL that exports a class has to handle that class differently than a DLL or EXE that uses the exported class.

The help has this to say about `LINK`:

LINK	Names a file to add to the link list for the current project.
Linkfile	A string constant naming an file (without an extension <code>.OBJ</code> is assumed) to link into the project. Normally, this would be the same as the parameter to the <code>MODULE</code> attribute (which by default is a source file), but may explicitly name a <code>.LIB</code> or <code>.OBJ</code> file.
Flag	A numeric constant, equate, or Project system define which specifies attribute as active or not. If the flag is zero or omitted, the attribute is not active, just as if it were not present. If the flag is any value other than zero, the attribute is active.

Let's say you're creating a hand coded DLL that will contain all your generic classes (easy to do, I assure you - just wait a bit). In that case you'll want the `LINK` attribute to be active, because you want the class source to be compiled and linked.

If you're using that class in another app, one that will make use of the DLL with the class source compiled in,

you want `LINK` to be inactive, because your app is going to make calls directly into the generic class DLL. You don't want to link in the source.

This is why you'll see `_ABCLinkMode_` defined as `true` (1) in an app that exports the ABC classes (typically also a data DLL), and defined as 0 in an app that makes use of that base class/data DLL.

But it's not enough to take care of the linking differences. There's also the `DLL` flag, about which the help says this:

DLL	Declares a variable, FILE, QUEUE, GROUP, or CLASS defined externally in a .DLL.
flag	A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the flag is zero, the attribute is not active, just as if it were not present. If the flag is any value other than zero, the attribute is active.

The help also notes that "The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the variable."

This translates loosely as "If you're using a class exported from a DLL, and you don't have an active DLL attribute on the class, your app is gonna blow chunks."

The problem I have is I can never seem to keep `_ABCLinkMode_` and `_ABCDllMode_` straight in my head. Objectively I know that a DLL that exports the ABC classes needs the defines

```
_ABCDllMode_=>0;_ABCLinkMode_=>1
```

while a DLL or EXE that uses that DLL with the exports needs the defines

```
_ABCDllMode_=>1;_ABCLinkMode_=>0
```

but it doesn't matter. It's only two variables, each with just two states, but my eyes glaze over anyway.

What complicates this further is that in my own code I don't want to use `_ABCLinkMode_` and `_ABCDllMode_`, because my classes are potentially useful in different applications (not just APP files). If I have two different applications, each made up of a data DLL, non-data DLLs, and one or more EXES, and I add a new class or method to my class library (which happens all the time), I now have to update the exports for each of those data DLLs. I don't want to do this - I want a single point of maintenance for my classes.

Third party vendors have the same problem - they don't want you to have to export their classes from your data DLL (the one that typically exports ABC classes). So they set up their own equates, which have to have their own project defines to ensure that their classes are handled correctly. I've seen apps with numerous pairs of DLL and LINK mode defines as needed by the various third party products.

This, not to put too fine a point on it, is nuts. In most circumstances there's absolutely no need for *any* project defines in any app *except* the one that compiles and exports the class(es).

I've only touched on two uses of the `LINK` and `DLL` attributes - linking classes into a DLL, and using classes exported from such a DLL. There are other possibilities such as linking in classes in an OBJ or LIB, but these aren't very common. I strongly suspect that almost everyone who reuses classes does so by exporting from one DLL and then using those exported classes in other DLLs and EXEs. This is the scenario I'm addressing.

## A better way

The DCL uses what I like to think is a much better way. It's certainly a lot simpler and less error-prone.

All DCL classes use a DCL-specific set of LINK and DLL attributes. For example:



```
DCL_System_String CLASS,TYPE,MODULE('DCL_System_String.CLW')|
    ,LINK('DCL_System_String.CLW',_DCL_Classes_LinkMode_)|
    ,DLL(_DCL_Classes_DllMode_)
```

Nothing unusual there. But wait! All DCL class INC files also have this statement at the top:

```
include('DCL_IncludeInAllClassHeaderFiles.inc'),once
```

That file looks like this:

```
OMIT('***',_Compile_DCL_Class_Source_)
_DCL_Classes_LinkMode_ equate(0)
_DCL_Classes_DllMode_ equate(1)
***
COMPILE('***',_Compile_DCL_Class_Source_)
_DCL_Classes_LinkMode_ equate(1)
_DCL_Classes_DllMode_ equate(0)
***
```

The key point is that in the Clarion project system if a variable is undefined it is assumed to have a value of zero. If `_Compile_DCL_Class_Source_` is never set, then the first pair of equates applies; if `_Compile_DCL_Class_Source_` is set to true then the second set applies.

Now there's just *one* equate that needs to be added to a project, and it *only* needs to be added if you actually want to compile the classes; otherwise it's assumed that you're using the classes as compiled into the DCL DLL.

## The easiest way

It's still a hassle typing in LINK and DLL attributes when creating a class for the first time. Or it used to be. I now use John Hickey's brilliant ClarionLive! Class Creator, part of the [ClarionLive utilities pack](#). Just set up a set of INC and CLW base class files, and Clive (as I like to call him) will do the rest. Fantastic!

## Summary

For years I've found the `DLL` and `LINK` attributes to be a huge pain, and on more than one occasion I've either forgotten to add them or added them incorrectly, with predictably ugly results.

The technique I've described in this article eliminates the need to set project defines to use exported classes. It takes a little more setup (which I've now completely automated using Clive), but only requires a single compilation symbol and that only for the DLL that exports the class. I can use my exported classes in any app without being concerned at all about whether I have the correct project defines set up.

[Like](#) Be the first to like this

No labels

[Pages](#) / [Clarion Content Home](#) / [The ClarionMag Library \(CML\)](#)

## Rebranding the DCL to the CML, and other changes

Added by David Harms, last edited by David Harms on Apr 16, 2014

With the move back to the ClarionMag brand, I realized I'd better rebrand the site's signature library, until now known as the DevRoadmaps Clarion Library (DCL), as the ClarionMag Library (CML).

### ClarionMagLibrary on GitHub

You can find the ClarionMagLibrary on GitHub at <https://github.com/clarionmag/ClarionMagLibrary>. This is the new home of this library - the DevRoadmapsClarionLibrary on GitHub will remain online for now but will not be updated.

To begin with I simply made a copy of the entire DCL development directory tree. I then went in to the libsrc directory and issued a

```
ren DCL_* CML_*
```

command, which worked only because I maintained the same number of characters at the start of the name.

I then loaded up all of the text files in that directory with Notepad++ and did a similar search and replace across all files (I could have used Clarion, but I like NPP for bulk operations). There were 1751 occurrences, so definitely this isn't something I'd want to do by hand.

I went through the same process with the templates. I was a bit worried about the template changes, because I have a lot of test apps that use the DCL templates and classes, and if I renamed the templates (not just the template files) I'd have to recreate all of the test apps from TXAs. But it turns out I didn't use the DCL\_ prefix in template names, so all I really need to do is unregister the old templates and register the new ones.

And of course I had to rename all of the unit test apps. I always do this by opening the app and doing a Save As to another name. I have a vague primal memory of once doing this with a DOS rename and having everything go south because none of the internal names were changed, e.g. for the target. Thinking it might be safe to do so now, I experimented. And no, the target name is still not changed if you do a DOS rename, and my paranoia is probably justified.

But it will take some time to go through all of the unit test apps, so for now I've left them out of the repository. They'll be back soon.

### Renaming the library DLL and LIB to CMgLib

When I first did the conversion I renamed the DevRoadmapsClarion project to ClarionMagLibrary. But that means that to ship product using the library you need to include a file called ClarionMagLibrary.dll with your product. Although I'd like all Clarion devs to know about and use the ClarionMag library, I'm not trying to advertise to everyone's clients. So I decided to make the actual DLL/LIB name a bit more cryptic. I thought about "CMLib" but a Google search for CMLib.DLL resulted in over 15,000 hits.

A Google search for CMagLib.DLL, however, yielded just two hits, only one of which was an actual match. So the new project name for the library is CMagLib.

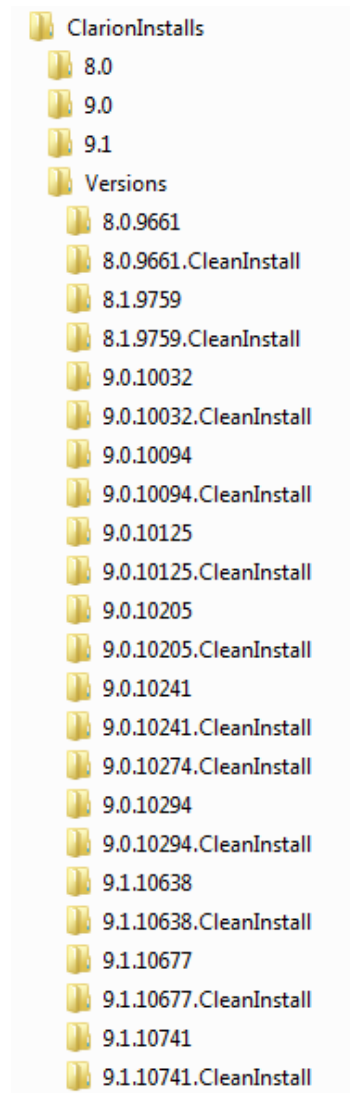
Renaming a hand coded project isn't all that difficult. All the file names need to be reloaded, and then you need to load up the project and solution files and do a search and replace. In my case I also needed to do a search

and replace for the files used by the CreateExportFile utility, which automatically generates the EXP file for all the classes exported by the library.

### Compiling for multiple Clarion versions

Although the need for version-specific recompiles declined with Clarion 8, there are times when you need an updated or older version of a library.

For a long time now I've kept multiple installs of Clarion on my machine. I also keep two copies of each install - a clean install, and one configured the way I usually use Clarion. As well, I typically only use the latest version of any given release, so my directory tree looks like this:



The real question is how do I create builds of the library for each release, or at least each release that needs an updated build?

One factor strongly in favor of automating the process is the library itself is a hand coded project, so I don't need to worry about keeping multiple versions of APPs around. The .cwproj files can be compiled in any release of Clarion from 8 (and probably 7, although I don't currently have it installed) onwards.

After some experimenting, I settled on the following batch files.

#### CompileCMagLibrary.cmd

```
call CompileForClarionVersion versions\8.0.9661
call CompileForClarionVersion versions\8.1.9759
```

```
call CompileForClarionVersion versions\9.0.10294
call CompileForClarionVersion versions\9.1.10741
call CompileForClarionVersion 9.1 baseline
```

CompileCMagLibrary.cmd uses a naming convention that follows my ClarionInstalls directory structure, and passes in the version folder name.

CompieForClarionVersion.cmd is a bit more complicated:

```
@echo off
if not exist f:\ClarionInstalls\%1\bin (
    echo.
    echo The Clarion release directory f:\ClarionInstalls\%1\bin could not be found
    echo.
    exit /b 1
)
echo.
echo Building ClarionMagLibrary for release %1
echo.
echo Compiling CMagLib.cwproj
echo.
call CompileCWProj.cmd CMagLib.cwproj f:\ClarionInstalls\%1\bin
echo Compiling CreateExportFile.cwproj
echo.
call CompileCWProj.cmd CreateExportfile.cwproj f:\ClarionInstalls\%1\bin
if "%2"=="baseline" (
    set BINFOLDER=..\bin
    set LIBFOLDER=..\lib
) else (
    set BINFOLDER=..\bin\%1
    set LIBFOLDER=..\lib\%1
)
echo.
echo BINFOLDER %BINFOLDER%
echo LIBFOLDER %LIBFOLDER%
echo.
echo Copying binary files...
echo.
if not exist %BINFOLDER% mkdir %BINFOLDER%
copy CMagLib.dll %BINFOLDER% /y
copy CreateExportFile.exe %BINFOLDER% /y
if not exist %LIBFOLDER% mkdir %LIBFOLDER%
xcopy ..\build\obj\debug\CMagLib.lib %LIBFOLDER% /D /Y
xcopy ..\build\obj\release\CMagLib.lib %LIBFOLDER% /D /Y
```

There are a few little tricks here. I'm setting a destination folder for the bin and lib files that matches the version used to compile the library, and I'm creating the folder if it doesn't exist. Also I'm using xcopy to bring the latest version of the library over, which may be the debug version or the release version. I don't want to copy an earlier debug version over a later release version or vice versa.

I still want a final compile that puts the latest version of the library in the default bin and lib directories. The "baseline" flag takes care of that.

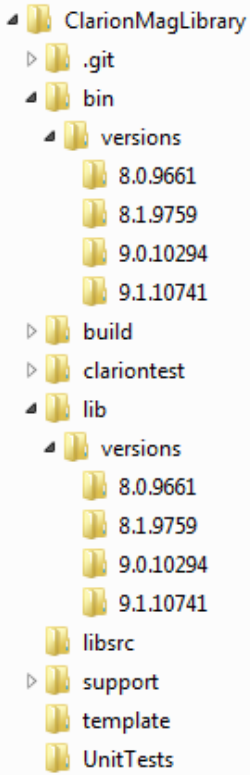
Finally, there's CompileCWProj.cmd:

```
@echo off
if "%1"==" " (
    echo You must specify a project as the first parameter
    exit /b 1
)
if "%2"==" " (
    echo You must specify a Clarion bin directory as the second parameter
    exit /b 1
)
if not exist %1 (
    echo Cannot find the project: %1
    exit /b 1
)
if not exist %2 (
    echo Cannot find the Clarion bin directory: %2
    exit /b 1
)
"C:\Windows\Microsoft.NET\Framework\v2.0.50727\msbuild" %1 /p:vid=min;line_numbers=True
/property:ClarionBinPath="%2"
if %errorlevel% neq 0 goto error
exit /b 0

:error
@echo CompileClarionProject.cmd returning error 1 on errorlevel %errorlevel%
exit /b 1
```

The meat is near the end, where the command line compile is invoked via MSBuild. Note the `/p:vid=min;line_numbers=True` option which tells the compiler to include the minimum amount of debugging info to allow for easy tracking down of GPFs when using the debug version of CLARun.dll. Given this option I don't really need the xcopy trick above, but if at some point I switch to a non-debug compile I'll have saved myself some grief.

Here's the ClarionMagLibrary directory tree with the bin and lib nodes expanded:



If you need a different version of the library, please post a comment below and I'll add it to the build process.

I'll be posting more on the code available inside the CML soon - stay tuned.

[Like](#) Be the first to like this

No labels

[Pages](#) / [Clarion Content Home](#) / [The ClarionMag Library \(CML\)](#)

## The ClarionMag Library (CML) on GitHub

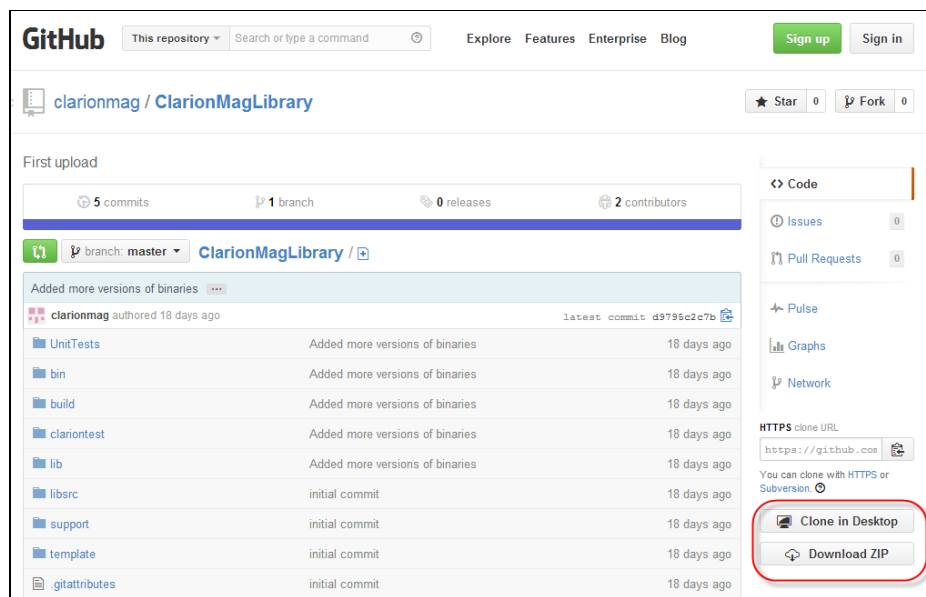
Added by David Harms, last edited by David Harms on May 13, 2014

### DCL to CML changes under way

We've copied the DCL docs to this new location and are still in the process of scrubbing the pages and fixing all the text and links. If you notice anything amiss feel free to please post a comment at the bottom of the page. Thanks!

The ClarionMag Library (CML) is available [by download from GitHub](#).

There are several ways you can get the library, but the two easiest are to clone the repository using GitHub for Windows or download the library as a zip file:



The screenshot shows the GitHub repository page for 'clarionmag / ClarionMagLibrary'. The repository has 5 commits, 1 branch, 0 releases, and 2 contributors. The latest commit is 'Added more versions of binaries' by clarionmag, authored 18 days ago. The repository contains several files and folders, including UnitTests, bin, build, clariontest, lib, libsrc, support, template, and gitattributes. The 'Clone in Desktop' and 'Download ZIP' buttons are highlighted with a red circle.

### Downloading the zip

Downloading the zip is dead easy - just click on the Zip button. But you won't get any automatic notifications of changes, and you won't be able to see what those changes are unless you download the zip again, extract it to a new location, and use a comparison tool to view the differences.

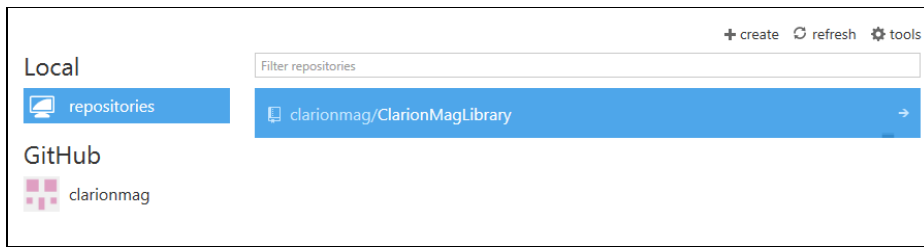
Cloning the repository with GitHub for Windows solves those problems.

### Cloning the repository

Dealing with Git repositories in Windows can be a royal pain. Happily, GitHub for Windows (released May 2012) makes Git a whole lot easier to use.

If you don't have GHfW installed, go to <http://windows.github.com/> and click the download link. When you run GHfW for the first time it will ask you to log in to GitHub.com. If you don't have a login click on the link in GHfW to register. After you've registered, log in using GHfW.

At that point you can go to the CML page at <https://github.com/clarionmag/ClarionMagLibrary> and click on the Clone in Windows button. GHfW will clone the CML repository, and you should see something like this in GHfW:



You may want to check out the June 8 2012 [ClarionLive webinar](#) recording, where I went through the process with Arnold and Lisa. **NOTE: At that time this site was still called DevRoadmaps and the library was the DevRoadmaps Clarion Library (DCL), so be sure you download the current ClarionMag Library (CML) instead.**

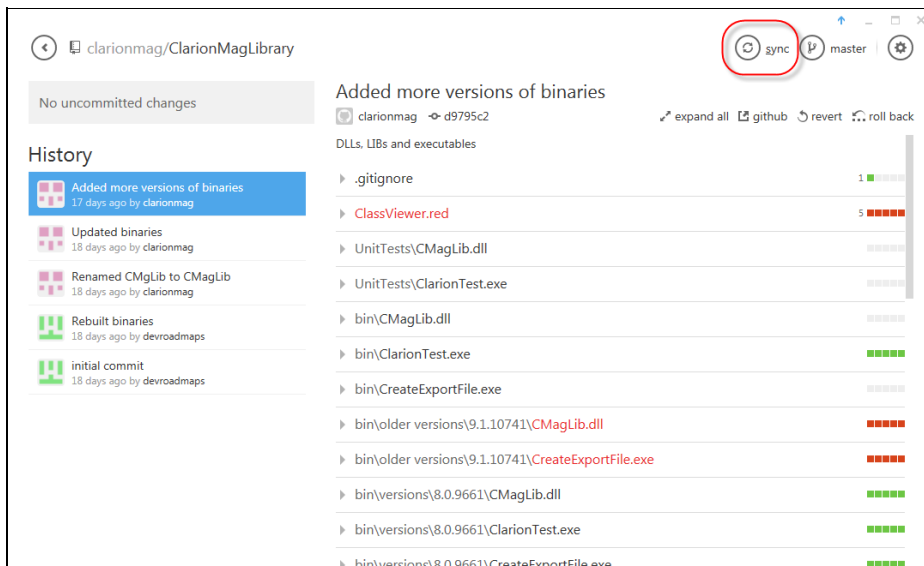
## About repositories

So here's the thing about Git repositories. Git is a distributed version control system, meaning that there doesn't have to be just one repository somewhere that everyone uses. Git is capable of synchronizing between repositories.

When you clone the CML you get the repository with all of its history. There's a good chance that will be a one-way street - when the CML gets updated, you'll use GHfW to synchronize your copy with the GitHub repository. But you can also make changes to your own code, commit those changes to your local repository, and take advantage of Git's ability to merge the GitHub changes with your local repository commits. Of course if you do go down this route you'll want to build any needed binary files yourself.

## Getting CML updates

Getting library updates is easy. Just run GHfW and bring up the CML repository. If there are pending updates the sync button will be enabled - click on it to bring your local copy up to date.



## Moving the repository

Chances are GHfW has put CML under your documents folder, and you probably want to move it to somewhere more Clarion-friendly.

To move a repository just move the entire ClarionMagLibrary folder tree to its new location. GHfW will notice that it can no longer find the files. Using Windows Explorer, drag and drop the ClarionMagLibrary folder onto



the GHW application. That's all you need to do.

## Using the library

Once you have the CML where you want it, you're ready to [start using it](#).

[Like](#) Be the first to like this

No labels