Apple IIGS
#86: Risking Resourceful Code

Revised by: Matt Deatherage March 1991
Written by: C.K. Haun <TR> September 1990

This Technical Note covers considerations you need to keep in mind when using code resources.
Changes since September 1990: Now lists XCMD and XFCN resources as "Apple's code" and notes that other restrictions apply to them as well.

Code resources are wonderful things that can make your life better than it ever was before. Code resources are necessary when writing CDevs and can be very useful for control definition procedures, code modules for extensible programs like resource editors—in fact, almost anywhere where you use regular compiled and linked code. But to do it right, you need to keep some rules in mind.

Apple's Code, Apple's Rules

The first code resources covered are the ones defined as fully supported by the System Software. These are `rCtlDefProc` ($800C), `rCodeResource` ($8017), `rCDEVCode` ($8018), `rXCMD` ($801E) and `rXFCN` ($801F). Before looking at the specifics, this Note describes in general terms what happens when the Resource Manager loads a code resource.

When you call the Resource Manager with a request for a code resource (or when the system does, as with `rCtlDefProc`s), it loads it like a normal resource. The Resource Manager finds the resource in a resource map in the current search path, allocates a handle for the resource using the attributes in the resource attribute bits, and loads the resource into memory.

Now the Resource Manager examines the `resConverter` bit in the resource header. If this bit is set, indicating that this resource needs to be converted (as it should be for an `rCtlDefProc`), the Resource Manager checks its tables to see if a resource converter has been logged in (with the `ResourceConverter` call). For code resources, the correct converter has been logged in by the manager associated with that resource type. For example, the Control Manager logs in the code resource converter for `rCtlDefProc` resource type.

For code resources, `InitialLoad2` is used to load the OMF from memory. Then the Resource Manager returns a handle containing a pointer to the start of the loaded, relocated code.

Rule 1: Code resources must be smaller than 64K

The code resource converter uses the `InitialLoad2` function of the System Loader to load and convert code resources. That means that code resources are restricted in the same way that loading from memory is. One of these restrictions is that the code must be 64K or less.

Rule 2: Compiled and linked code only

Again, since `InitialLoad2` is used to convert the code resource, the data must be in OMF format since `InitialLoad2` expects to relocate standard load segments. When you prepare your code for inclusion in a code resource, compile and link the code as you normally would for a stand-alone program. Use the file produced by the linker for inclusion in your resource fork. You can use Rez to move the code from a data fork to your application's resource fork with a line in your resource description file similar to the following:

```
read rCodeResource (MyCodeIDNumber,locked,convert)  "MyCompiledAndLinkedCode";
```

Rule 3: One segment please

Multiple segments are theoretically possible with code resources, but you have to manage memory IDs and the memory that the additional code segments use yourself. Since the code resource converter calls `InitialLoad2`, it uses the Memory Manager ID for the current resource application, and you cannot specify a different user ID directly. By changing your current resource application ID (by making an additional call to `ResourceStartUp` with a modified master ID, for example) you could manage multisegment code resources.

Rule 4: No dynamic segments

The dynamic segment mechanism does not work with code resources. Of course, your application can still use dynamic segments, but not code resource dynamic segments.

Rule 5: Set the right attributes

There are two sets of attributes you need to be concerned about for a code resource. The first set includes the standard resource attributes; the second set covers the attributes that the code itself has in the OMF image.

You need **both** sets to get the functionality you want. The resource attributes determine how the Resource Manager handles the resource. The OMF attributes control what `InitialLoad2` does when it converts your code from OMF in a resource handle to relocated executable code.

Remember, you need to set **both** sets of attributes.

The resource attributes you need to set are `locked` and `convert`. The `locked` flag is necessary to prevent the resource from moving while `InitialLoad2` processes it, and the `convert` flag is needed to signal the Resource Manager to call the code resource converter.

You must set the static OMF attribute, the others (like no special memory) you set as appropriate for your code in your application.

Rule 6: Know where to go

The handle you get back from the Resource Manager when you load and convert a code resource points to the beginning of the relocated and ready-to-execute code, **not** to the image of the code that is stored in the resource fork. So you can immediately jump to this code to execute it.

You can override this if you like—clear the `resConverter` bit in the resource attributes. If this bit is zero, the Resource Manager does not call any resource converter (including the code resource converter).

Rule 7: Remember the Write

Keep in mind that any resource that uses a converter uses that converter both for reading and writing the resource. If you write out a code resource, the Resource Manager calls the `Write` routine for the code resource converter, which currently writes without doing any conversion—it does **not** reconvert the code in memory back to OMF format. However, some converters (perhaps one you write) could reconvert the resource before writing it out.

Your Code, Your Rules

If you want to define your own code resource type (with a resource type of less than `$8000` and greater than 0) you may want to follow the same rules as the system code resources use. In fact, you can even use the same code resource converter, by using the `ResourceConverter` call with your resource type, and log the code resource converter as the converter to use with your resource type, like the following:

```
       pha
       pha                     ; return space
       _GetCodeResConverter    ; Misc Tools call to return the loader relocation code
pointer
*                              ; (leave it on the stack for the next call)
       pea $0678               ; resource type you want to convert with this
converter, any
*                              ; Application type you wish
       pea %01                 ; add this converter to the Application converter list,
*                               ; and log this routine in
       _ResourceConverter
```

or you can do whatever you like with the resource, including not having a converter and doing all the relocation and memory management of the code yourself. This can give you the ability to add more functionality than the standard code resources provide—dynamic segmentation is one feature you could implement if you want to handle all the details yourself.

Or, you can manage the code any way you want, but keep the built-in system functions in mind, and use as many of them as you can. Make your life simpler.

One Final Note

If one of your resources is marked `convert` and `preload` the Resource Manager **only** preloads that resource if the converter for that resource is logged in as a converter for that type. If the Resource Manager cannot find the converter, it does not preload the resource.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 3
- *GS/OS Reference*
- *HyperCard IIGS Script Language Guide*
- HyperCard IIGS Technical Note #1, Corrections to the *Script Language Guide*
- Apple IIGS Sample Code #9, Lister