New Technical Notes

Macintosh

Developer Support

QT 05 - Component Manager version 3.0

QuickTime

Written by: Jim Reekes & Dave Radcliffe March 1994

This note contains information regarding the version of the Component Manager that shipped with QuickTime 1.6 and the changes necessary to support native PowerPC components.

Topics

- QuickTime 1.6 and Component Manager 3.0
- **Native PowerPC components**

Introduction

The Component Manager in QuickTime 1.6.x and for the Power Macintosh (PowerPC) release has some new features. It has added the ability to automatically resolve conflicts between different versions of the same component. It will ensure that only the most recent version of a given component is actually registered. The Component Manager now supports Icon Suites for a component, so a component's icon no longer has to be just black and white. In addition, the Component Manager can support code written in the native format of the PowerPC.

The result returned for the Gestalt selector gestaltComponentMgr will be 3, indicating version number 3 of the Component Manager. This is the version being discussed in this note. To insure that you have the features discussed here, check that version 3 is installed.

For support of the Power Macintosh, the Component Manager has been extended to allow use of native PowerPC components. When the Component Manager loads a native component on the Power Macintosh, it calls uses the Code Fragment Manager and calls GetMemFragment and then later CloseConnection when it unloads your code resource (specified in a ComponentPlatformInfo). This is how the Component Manager supports a native code fragment.

A component can support multiple platforms such as the 68K and PowerPC. Existing 68K code is always supported on the Power Macintosh through emulation. But you can also have native PowerPC code for your component to support better performance. The Component Manager will allow you to create a component that contains both code formats, so that you can support all platforms with a single component. The Component Manager also was

extended in a way that allows for native PowerPC only components (without any 68K code support).

Extended ComponentResource

The ComponentResource data structure (the 'thng' resource) has been extended. These extensions define additional information about the component. The complete data structure is shown below. The first portion is the same as the existing ComponentResource, with the new

fields added at the end. The Component Manager determines if it is present by examining the size of the 'thng' resource.

```
struct ExtComponentResource {
      ComponentDescription   cd;                    /* Registration parameters */
      ResourceSpec component;                       /* resource where Component code is found */
      ResourceSpec componentName;                   /* name string resource */
      ResourceSpec componentInfo;                   /* info string resource */
      ResourceSpec componentIcon;                   /* icon resource */

      // new data for Component Manager version 3
      long              componentVersion;           /* version of Component */
      long              componentRegisterFlags;     /* flags for registration */
      short                   componentIconFamily;  /* resource id of Icon Family */
      long              count;                      /* elements in platformArray */
      ComponentPlatformInfo platformArray[1];
};
```

componentVersion
The componentVersion field contains the version number of the component. This should be identical to the value returned by GetComponentVersion. For convenience, if this value is set to 0, the component is called to get the version. This is useful during development. The version number stored in the ComponentResourceExtension is used by the Component Manager to avoid having to load and call the component to retrieve the component's version during startup.

componentRegisterFlags
The componentRegisterFlags allow you to define additional register information. These flags are discussed below.

```
/* Component Resource Extension flags */

componentDoAutoVersion                  = (1<<0)
componentWantsUnregister                = (1<<1)
componentAutoVersionIncludeFlags              = (1<<2)
componentHasMultiplePlatforms           = (1<<3)
```

The componentDoAutoVersion flag tells the Component Manager that you want your component registered only if there is no later version available. If there is an older version of the component installed, it will be unregistered. If an older version of the same component attempts to register after you, it will be immediately unregistered. Further, if a newer version of the same component registers after you, you will automatically be unregistered. Using the automatic version control feature of the Component Manager allows you to make sure that only the most recent version of your software is running on a given machine, regardless of how many versions may be installed.

The componentWantsUnregister flag indicates that your component wants to be called when it is unregistered. This is useful if your component allocates global memory at register time, for example. The prototype of the unregister message is identical to the register message. If your component has never been opened, its unregister message is not be called. The routine selector for unregister is given below.

```
kComponentUnregisterSelect              = -7
```

The componentAutoVersionIncludeFlags flag tells the Component Manager to use the component flags as criteria for its component search. If a component wants automatic version control, the Component Manager has to search for similar components. Normally, the Component Manager searches only for another component using the type, subType, and manufacturer fields of a ComponentDescription record. This flag tells the Component Manager to include the componentFlags in its search.

The componentHasMultiplePlatforms flag indicates that your component contains multiple versions of the code for different platforms. If you plan on supporting the PowerPC native code format, then you need to use the ComponentPlatformInfo within the component resource structure. Then set this bit in the componentRegisterFlags field. If this bit is not set then the code is assumed to be 68K format. Without this flag being set, the Component Manager will ignore any ComponentPlatformInfo.

componentIconFamily
Finally, the componentIconFamily field allows you to provide the resource ID of a System 7 Icon Suite. If this field is 0, it indicates that there is no icon suite.

count
This is the number of elements contained in the ComponentPlatformInfo array.

platformArray
This is an array of elements that describe the code to be used for different platforms. If the platform is for 68K, then the information within this element is a copy from the componentFlags of the ComponentDescription and ResourceSpec of the original ComponentResource structure. This insures backwards compatibility with older Component Managers. If the component contains native code support for the PowerPC, then an element of the array will contain the information about its componentFlags, resource type, and resource ID.

The platformType field is a value that represents which platform the component code is to support. The Gestalt result for selector gestaltSysArchitecture will be matched with the value in platformType of the ComponentResource. If a match is found, then that code is used to support the given platform.

```
    gestalt68k          = 1,                    /* Motorola MC68K architecture   */
    gestaltPowerPC      = 2,                    /* IBM PowerPC architecture      */

struct ComponentPlatformInfo
{
    long            componentFlags;             /* flags of Component */
    ResourceSpec    component;                  /* resource where Component code is found */
    short           platformType;               /* gestaltSysArchitecture result */
};
```

Component Manager version 3 routines

GetComponentIconSuite

GetComponentIconSuite returns an Icon Suite for the given component. This call works only under System 7 or later. If called on System 6, it returns an error. If the component doesn't have an Icon Suite but does have a Component Icon (as returned by GetComponentInfo), GetComponentIconSuite creates an Icon Suite containing just the black-and-white Component Icon. In this way, you can use GetComponentIconSuite whether or not a component has an Icon Suite.

```
pascal OSErr GetComponentIconSuite(Component aComponent, Handle *iconSuite)
```

aComponent Component ID, retrieved with FindNextComponent.
iconSuite Pointer to the Icon Suite you will receive.

RegisterComponent
RegisterComponentResource
RegisterComponentResourceFile

The only change made to these routines was to modify the use of the global parameter. The upper byte now contains the platform ID to be used by the component being registered. This change is necessary because these calls do not have access to the ComponentResource which contains the ComponentPlatformInfo. If the upper byte of the global parameter is zero, then the platform is assumed to be the platform68k.

How to create a PowerPC ComponentResource

The basics step for running on a Power Macintosh with a native component are:

- Create component code fragment with native PowerPC code
- Main entry point to code is a mixed mode routine descriptor
- Package component code fragment as a resource
- If you supply an interface for the component to be called directly, then for PowerPC code to call your component you must provide custom glue to make the call.
- Create the extended 'thng' resource using the ComponentPlatformInfo

Each of these steps are discussed in more detail below:

Creating the component code fragment

The first step in creating a native PowerPC component is to port your code. For complete details on porting to PowerPC, see *Inside Macintosh: PowerPC System Software.* Especially important for the following discussion is an understanding of the Mixed Mode and Code Fragment Managers.

Like other code ported for PowerPC, anytime your code uses a callback function (ProcPtr), it must be converted to a UniversalProcPtr. But unlike callbacks defined by the system, callbacks to your component have their own function prototypes. With the exception of some callbacks defined for QuickTime components, there are no system supplied function prototypes or UniversalProcPtrs, so you must create these yourself.

If, in response to a request code, your component dispatches to internal functions using CallComponentFunction or CallComponentFunctionWithStorage, then this is a place where you must use a UniversalProcPtr.

Suppose your component currently responds to an open request as follows:

```
switch (params->what)
{
    case kComponentOpenSelect:          /* Open request */
    {
        result = CallComponentFunctionWithStorage (storage, params, MyOpen);
        break;
    }
```

MyOpen is an internal function callback, so you must create a RoutineDescriptor/UniversalProcPtr for it. MyOpen is declared as follows:

```
pascal ComponentResult MyOpen (Handle storage, ComponentInstance self);
```

The first step is to create a ProcInfo value for this function:

```
enum {
    uppMyOpenProcInfo = kPascalStackBased
        | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
        | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Handle)))
        | STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(ComponentInstance)))
};
```

Next you must update your source to build a UniversalProcPtr and use it. You could use NewRoutineDescriptor for this purpose, but the disadvantage is that creates a heap object which your component must dispose of properly.

An alternate approach is to declare a global RoutineDescriptor (global variables are not a problem for a native PowerPC component, since a code fragment automatically has global variables):

```
#ifdef powerc
RoutineDescriptor MyOpenRD = BUILD_ROUTINE_DESCRIPTOR (uppMyOpenProcInfo, MyOpen);
#endif
```

If you want your code to be compilable for both 68K and PowerPC, using the Universal Interfaces, then to avoid a lot of conditional compilation, the following macros may be useful:

```
#ifdef powerc
#define CallComponentFunctionWithStorageUniv(storage, params, funcName) \
    CallComponentFunctionWithStorage(storage, params, &funcName##RD)
#define CallComponentFunctionUniv(params, funcName) \
    CallComponentFunction(params, &funcName##RD)
#define INSTANTIATE_ROUTINE_DESCRIPTOR(funcName) RoutineDescriptor funcName##RD = \
    BUILD_ROUTINE_DESCRIPTOR (upp##funcName##ProcInfo, funcName)
#else
#define CallComponentFunctionWithStorageUniv(storage, params, funcName) \
    CallComponentFunctionWithStorage(storage, params, (ComponentFunctionUPP)funcName)
#define CallComponentFunctionUniv(params, funcName) \
    CallComponentFunction(params, (ComponentFunctionUPP)funcName)
#endif
```

These macros, exactly analogous to CallComponentFunction and CallComponentFunctionWithStorage, generate the appropriate code when compiled for 68K and PowerPC. Note that the PowerPC macro expansion depends on the global RoutineDescriptor name being FuncNameRD, i.e. the name of the function with RD appended. The INSTANTIATE_ROUTINE_DESCRIPTOR macro can be used for that purpose:

```
#ifdef powerc
INSTANTIATE_ROUTINE_DESCRIPTOR(MyOpen);
#endif
```

This is identical to the declaration of MyOpenRD earlier, but simplifies the editing.

With all the conditional stuff out of the way, then the original code can simply be updated by replacing CallComponentFunctionWithStorage with CallComponentFunctionWithStorageUniv:

```
    switch (params->what)
    {
        case kComponentOpenSelect:          // Open request
        {
            result = CallComponentFunctionWithStorageUniv(storage, params, MyOpen);
            break;
        }
```

Repeat the above steps for all internal component dispatches you make.

Setting the main entry point

Lastly, you must set up the entry point into your component correctly. Unlike a 68K code resource, a PowerPC code fragment (which your component will be) has a well defined entry point. The Component Manager, rather than just jumping to the start of the code resource, will call the main entry point, as defined when linking, instead.

But the Component Manager is 68K code, which means your main entry point must be a RoutineDescriptor. You can set that up as follows:

```
pascal     ComponentResult   main       (ComponentParameters  *params,
                                                Handle                          storage);
#ifdef powerc
enum {
      uppMainProcInfo = kPascalStackBased
            | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
            | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(ComponentParameters *)))
            | STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(Handle)))
};

RoutineDescriptor MainRD = BUILD_ROUTINE_DESCRIPTOR(uppMainProcInfo, main);
#endif
```

When you link the component, you must then specify MainRD as the entry point.

Note: Your development environment may issue a warning because your main entry point is in a data section, not a code section. You may ignore the warning.

Note: If your code is dependent on C or C++ runtime initializations, then your main entry point would be __start or __cplusstart, respectively, rather than main. Modify the previous example accordingly.

Note: Some components rely on a “fast dispatching” mechanism for calling component functions. This mechanism is dependent on the 68K architecture and is unsupported for native components, although it will work for emulated components running on the Power Macintosh.

Last Note: In all these modifications for PowerPC, the most difficult thing to get right is the ProcInfo value. It’s very easy to make a “cut and paste” error, or get a type wrong (short instead of short *). If your component is crashing the first thing to check (and check and check and check!) are the ProcInfo values.

Packaging the PowerPC component into a resource

PowerPC development tools create your PowerPC code in a code fragment in the data fork of the file. Your component code must be a resource (the resource type and id are specified in the 'thng' resource described below). You can use the MPW Rez “read” command to read from the data fork into a resource. For example:

```
read 'mycp' (130) mycomponent.pef;
```

reads the code fragment from the file mycomponent.pef and creates the resource 'mycp'(130).

Providing an interface to the component

If you wish your component to be called directly, you must also supply an interface so callers know how to call it. For standard functions, such as Open, Close, Version, etc., this is not a problem as the Component Manager supplies functions to do this for you. Nor is this a

problem if you are writing QuickTime components, as QuickTime supplies standard interfaces and libraries for calling components.

But one of the advantages of the Component Manager is it lets you define your own routines with their own parameter lists, and for these routines you must supply an interface. Typically, for 68K this involved providing callers an interface file with function prototypes for your calls and inline 68K assembly to actually make the call.

Obviously, the inline 68K code is a problem for a native PowerPC caller, so you must provide glue to accomplish the same thing. The following discussion also applies to calling a 68K component from PowerPC code. The interface is the same, either way.

To take the example for Inside Macintosh: More Macintosh Toolbox, page 6-30, you might have a call like:

```
pascal ComponentResult DrawerSetup (ComponentInstance myInstance, Rect *r) =
    ComponentCallNow (kDrawerSetUpSelect, 4);
```

ComponentCallNow is a macro that expands to inline 68K code that pushes additional parameters and then executes an A-trap to call the Component Manager.

The first thing when using the new Universal Headers, is that the definition of ComponentCallNow has changed slightly. The above declaration would change to:

```
pascal ComponentResult DrawerSetup (ComponentInstance myInstance, Rect *r)
    ComponentCallNow (kDrawerSetUpSelect, 4);
```

The only difference in this declaration is that the '=' character is missing. This is necessary to allow the code to compile for both 68K and PowerPC.

For 68K code, ComponentCallNow continues to expand to inline 68K code, but for PowerPC, the ComponentCallNow macro expands to nothing, which means the above declaration reduces to:

```
pascal ComponentResult DrawerSetup (ComponentInstance myInstance, Rect *r) ;
```

You must now supply glue for DrawerSetup that does the same thing on PowerPC as the 68K inlines would do.

The strategy here is to mimic what 68K code calling your component would do. Namely, push a bunch of parameters on the stack, then call the component. You do that by building a struct that looks like the parameters **as they would appear on the 68K stack**. Each call will require a different struct because each call can have different parameters.

Use the struct below (DrawerSetupGluePB) as a template. The first three fields, componentFlags, componentParamSize, and componentWhat are required, as is the last field, which is the component instance.

componentFlags is unused and should be zero.

componentParamSize is the size, in bytes, of the parameters to the call, not counting the component instance. This is the same number that is passed as the second parameter in a ComponentCallNow macro call, and should be the same as the size of the params struct, discussed below.

componentWhat is the selector for your component call. Its the same as the first parameter to a ComponentCallNow macro call.

The params field is a separate struct that exactly mirrors your parameters. This must be customized for your call. A separate struct is used here because it simplifies the sizeof calculation for the componentParamSize field. Parameters in this struct are specified in *reverse* order from the parameter list.

Note: Remember that the struct mirrors 68K stack alignment, *not* 68K struct alignment. This means that byte parameters, e.g. char or Boolean, get passed as two bytes, not one. The struct must mirror that fact, so you must declare byte fields to be a byte field followed by a pad byte field and take it into account in your parameter size calculations.

Once you have the struct, initialize it as shown in the example, and call the component via CallUniversalProc with the CallComponentUPP. CallComponentUPP is declared for you and is part of the InterfaceLib. You don't need to do anything special to use it.

uppCallComponentProcInfo should have been in the interfaces, because the call is always the same, but it's not, so it's defined below.

```
enum {
	uppCallComponentProcInfo = kPascalStackBased
		| RESULT_SIZE(kFourByteCode)
		| STACK_ROUTINE_PARAMETER(1, kFourByteCode)
};
```

Here's the code for the glue function. Once you have the structure defined, create an instantiation of it, and initialize it. Finally, call the component using CallUniversalProc as shown in the example.

```
pascal  ComponentResult  DrawerSetup (ComponentInstance myInstance, Rect *r)
{
#define kDrawerSetupParamSize    (sizeof (DrawerSetupParams))

#ifdef powerc
#pragma options align=mac68k
#endif
	struct DrawerSetupParams {
		Rect *theRect;			/* Your parameters go here!!  In reverse order from parameter list. */
	};
	typedef struct DrawerSetupParams DrawerSetupParams;

	struct DrawerSetupGluePB {
		unsigned char		componentFlags;		/* Flags - set to zero */
		unsigned char		componentParamSize;	/* Size of the params struct */
		short			componentWhat;		/* The component request selector */
		DrawerSetupParams	params;			/* The parameters, see above */
		ComponentInstance	instance;		/* This component instance */
	};
	typedef struct DrawerSetupGluePB DrawerSetupGluePB;
#ifdef powerc
#pragma options align=reset
#endif
	DrawerSetupGluePB myDrawerSetupGluePB;

	myDrawerSetupGluePB.componentFlags = 0;
	myDrawerSetupGluePB.componentParamSize = kDrawerSetupParamSize;
	myDrawerSetupGluePB.componentWhat = kDrawerSetUpSelect;
	myDrawerSetupGluePB.params.theRect = r;
	myDrawerSetupGluePB.instance = myInstance;
```

```
    return CallUniversalProc(CallComponentUPP,
                uppCallComponentProcInfo,  &myDrawerSetupGluePB);
}
```

Repeat the above steps for all the public functions for your component. To allow for future updating, the best way to make this glue available to your clients is to build the glue into a Code Fragment Manager shared library that is built into your component. Provide your client with an XCOFF file to link against. That way, if the glue changes, the client applications will not have to be relinked. Be sure you choose a unique name for the glue library to avoid possible name conflicts.

Creating the extended thng' ComponentResource

Here is how to create the 'thng' ComponentResource for a component that supports both platform68k and platformPowerPC. This is the source for MPW Rez using the latest version of Types.r that supports the UseExtendedThingResource template. Before using the new Types.r you need to define the UseExtendedThingResource conditional with the value 1. A component defined with this resource will work for all previous versions of the Component Manager. By keeping the original portions of the ComponentResource setup for the platform68k information, it allows your component to work on all 68K Macintosh computers. Adding the new information about your code fragment for the Power Macintosh allows the Component Manager for that machine to use your native code.

```
resource 'thng' (128, purgeable) {
  kComponentType,
  kComponentSubType,
  kComponentCreator,
  cmpWantsRegisterMessage,
  kAnyComponentFlagsMask,
  k68KCodeType, k68KCodeID,
  'STR ', kComponentNameStringID,
  'STR ', kComponentInfoStringID,
  'ICON', kComponentIconID,

#if UseExtendedThingResource
  0x00010001,                   /* version 1.1 */
  componentHasMultiplePlatforms,
  kComponentIconFamilyID,
  {
    cmpWantsRegisterMessage, k68KCodeType, k68KCodeID, platform68k,
    cmpWantsRegisterMessage, kPowerPCCodeType, kPowerPCCodeID, platformPowerPC
  };
#endif
};
```

If you have a component that only supports the 68K Macintosh, then you do not need to use the extended ComponentResource structure. However, if you wish to utilize Icon Families and automatic version registration, then use the extended ComponentResource without the ComponentPlatformInfo and do not set the componentHasMultiplePlatforms flag of the componentRegisterFlags. You may also include the ComponentPlatformInfo if you wish to and just have a single element that describes your 68K component code.

If you have a “fat” component, with both 68K and PowerPC code, set the component flags as you would for the 68K only case and duplicate that information in the ComponentPlatformInfo portion of the extended resource. That will allow your component to work correctly for versions of the Component Manager that are not aware of the extended 'thng' resource.

If you have a component that only supports the PowerPC in native mode, then you must use the extended ComponentResource. In this case, some care must be taken so that the component will not be registered on 68K machines. Set the ResourceSpec field in the non-extended part of the 'thng' resource to zero. In addition, set the component flags in the non-extended part of the resource to cmpWantsRegisterMessage, *regardless of whether or not you handle the register message.* This will cause the 68K Component Manager to attempt to register your component, it will fail, because there is no 68K code resource and your component will not be registered.

For the PowerPC case, you need to include a single ComponentPlatformInfo element that describes your PowerPC native component code for PowerPC implementations of your component to be registered. Set the component flags in the extended portion of the resource as you would normally.

Component Manager interfaces

```
/* MPW Rez interfaces */

#define cmpWantsRegisterMessage                 (1<<31)         /* bits for component flags */

#define componentDoAutoVersion                  (1<<0)          /* bits for registration flags */
#define componentWantsUnregister                (1<<1)
#define componentAutoVersionIncludeFlags        (1<<2)
#define componentHasMultiplePlatforms           (1<<3)

type 'thng' {
        literal longint;                                        /* Type */
        literal longint;                                        /* Subtype */
        literal longint;                                        /* Manufacturer */
        unsigned hex longint;                                   /* component flags */
        unsigned hex longint    kAnyComponentFlagsMask = 0;     /* component flags Mask */
        literal longint;                                        /* Code Type */
        integer;                                                        /* Code ID */
        literal longint;                                        /* Name Type */
        integer;                                                        /* Name ID */
        literal longint;                                        /* Info Type */
        integer;                                                        /* Info ID */
        literal longint;                                        /* Icon Type */
        integer;                                                        /* Icon ID */
#if UseExtendedThingResource
        unsigned hex longint;                                           /* version of Component */
        longint;                                                /* flags for registration */
        integer;                                                        /* resource id of Icon Family */
        longint = $$CountOf(ComponentPlatformInfo);
        wide array ComponentPlatformInfo {
                unsigned hex longint;                           /* component  flags */
                literal longint;                                /* Code Type */
                integer;                                                /* Code ID */
                integer platform68k = 1, platformPowerPC = 2;   /* platform type */
                };
#endif
};

/* MPW C interfaces */

enum {
#define gestaltComponentMgr             'cpnt'          /* Component Mgr version */

#define gestaltQuickTimeFeatures        'qtrs'      /* QuickTime features */
        gestaltPPCQuickTimeLibPresent = 0,      /* PowerPC QuickTime glue library is present */


#define gestaltSysArchitecture 'sysa'                   /* Native System Architecture */
        gestalt68k = 1,                         /* Motorola MC68K architecture */
```

```
    gestaltPowerPC = 2,                         /* IBM PowerPC architecture */

    /* componentRegisterFlags flags for ComponentResourceExtension */
    componentDoAutoVersion          = (1<<0),
    componentWantsUnregister        = (1<<1),
    componentAutoVersionIncludeFlags = (1<<2),
    componentHasMultiplePlatforms   = (1<<3)
};

struct ComponentPlatformInfo
{
    long            componentFlags;         /* flags of Component */
    ResourceSpec    component;              /* resource where Component code is found */
    short           platformType;           /* gestaltSysArchitecture result */
};
typedef struct ComponentPlatformInfo ComponentPlatformInfo;

struct ExtComponentResource {
    ComponentDescription   cd;              /* Registration parameters */
    ResourceSpec    component;              /* resource where Component code is found */
    ResourceSpec    componentName;          /* name string resource */
    ResourceSpec    componentInfo;          /* info string resource */
    ResourceSpec    componentIcon;          /* icon resource */

    // new data for Component Manager version 3
    long            componentVersion;       /* version of Component */
    long            componentRegisterFlags; /* flags for registration */
    short           componentIconFamily;    /* resource id of Icon Family */
    long            count;                  /* elements in platformArray */
    ComponentPlatformInfo platformArray[1];
};
typedef struct ExtComponentResource ExtComponentResource;
```

Further Reference:

- *Inside Macintosh: More Macintosh Toolbox* (Component Manager)
- *Inside Macintosh: PowerPC System Software* (Mixed Mode Manager and Code Fragment Manager)
- Macintosh Technical Note, Drawing Icons the System 7 Way (M.IM.IconDrawing).