VectorWorks SDK Manual

For VectorWorks 8.5

Diehl Graphsoft, Inc.
10270 Old Columbia Road
Columbia, MD 21046-1751
Telephone: 410-290-5114
Fax: 410-290-8050
www.diehlgraphsoft.com

COPYRIGHT 2000 Diehl Graphsoft, Inc. All Rights Reserved.

QuickTime and the QuickTime logo are trademarks used under license.

Documentation by: Andrew Campbell, Sean Flaherty, Steve Johnson, Jeffrey Koppi, Chris Nebel, and Paul Pharr. Layout by Sean Flaherty, John Ferdock, and Don Webster.

This document was illustrated using VectorWorks.

Important Disclaimer:

This document is substantially the same as that which was released with the MiniCAD 5.0 SDK with the exception that it does not contain the documentation for each callback function. That information is contained in the FileMaker database "Braque SDK Library.fp3" that is included with the downloadable SDK.

Many changes have been made to the SDK since this document was current - not the least of which is the port to Microsoft Windows. We will make an effort to bring this document up-to-date as soon as possible, but we are releasing this now in electronic form due to the numerous requests for the information contained within.

Please look at the following documents in the READ ME directory of the SDK for the most current information:

SDKInfo.txt
Release Notes.txt
SDK Plug-in Objects.html
SDK Undo.html

Paul C. Pharr
CAD Software Manager
Diehl Graphsoft, Inc.
January 20, 2000

Contents

INTRODUCTION **5**

Document Layout *5*
Included Files *5*
The SDK Future *5*

SECTION I: THE EXTERNAL ENVIRONMENT **6**

The External and the User *6*
The Definition of an External *7*
Tool or Menu Command? *8*
Menu Commands *8*
Menu Command Main Function *9*
Menu Command Actions *9*
Menu Command Definition Resource: MITM *12*
Tools *13*
Tool Main Function *13*
Tool Actions *14*
Tool Definition Resource: TDef *15*

SECTION II: THE VECTORWORKS ENVIRONMENT **17**

Basic Data Types *17*
Entities *17*
Lists *17*
Drawings *18*
List Traversal *18*
Graphic Entities *19*
Services *23*

SECTION III: THE DEVELOPMENT ENVIRONMENT **25**

Cross-Platform Development *25*
Project and Compiler Settings *25*
Managing A4 ..... *25*
Using Libraries in an External *26*
CallBackPtrs *27*
Global Data *27*
Debugging *28*

INDEX **29**

Introduction

VectorWorks defines an open architecture for tool and menu command development that allows developers to supplement or replace existing VectorWorks functionality. These new tools and menu commands are, from the user's perspective, indistinguishable from those built into VectorWorks. As such, they are first class solutions for the user. The opening of the command architecture in VectorWorks is made possible by the formalizing of the relationships between the user and tools and menus and between those tools and menus and VectorWorks. The purpose of this document is to define these relationships and describe how to implement a new tool or menu command from these specifications. Tools and menu commands that are developed using the SDK are referred to as "externals" or "plug-ins".

Document Layout

This document is divided into three sections. The first section defines and describes external tools and menu items and the environment they exist within. The second section describes how VectorWorks documents are organized and some of the services that the application provides to externals. The final section describes the peculiarities of developing externals in the development environment.

Included Files

In addition to this document, the SDK includes release notes, a function reference, and other documents. It also includes the source code and libraries necessary to build externals. Full source code for several sample externals is also provided.

The SDK Future

We have tried to create a flexible, expandable external environment to ease the creation of powerful additions to our product line. Diehl Graphsoft will continue to make improvements to the SDK and will try to always support old externals with our updates, but makes no promises of future compatibility. Use of programming techniques such as factoring (separating implementation and interface code) will ease the transition to new platforms and can be done more easily early in the development process than later.

Section I: The External Environment

The VectorWorks open environment is composed of 4 pieces: the user, the tools and menu commands, the drawing, and the VectorWorks application. The user works with the tools and menu commands to manipulate the drawing. VectorWorks provides the means of communication between the other parts as well as the essential functionality of a CAD system (coordinate systems, objects, math routines, etc.).

The system works because each part has a well-defined role and each part, except one, has a well-defined protocol to carry out its role. The one exception is, of course, the user, who will not submit to any protocol or regulation. The best that each part can do when dealing with the user is to have a clear purpose, to be tolerant of erroneous input (there is no such thing as a user error, only bad interfaces), and to provide information to the user in every way possible so that he or she will naturally make the right choices.

• The user's role is to generate events, from mouse movement and button clicks to requests for a wall or to rotate the world. The user is the action to which everything else reacts.
• The drawing is the data that the user creates. Its job is to exist.
• VectorWorks' role is the enabler. It provides the framework in which the other parts meet.
• The role of the external is to respond to the user, servicing his or her requests to create and manipulate the drawing, thereby building on the VectorWorks framework useful software. In less grand terms, a tool or menu command solves a user problem with the help of VectorWorks.

The External and the User

As stated above, the user will relate to the external in any way he or she pleases. An external, however, in dealing with the user, must provide the following:

• Functionality - This is not that difficult to do. If you are reading this document then you probably have some ideas, ranging from the just plain silly to the revolutionary. The difficult thing about functionality is making it useful. The rest of the requirements exist to help do this (though they are by no means sufficient).

• Undo - Documentation may guide the user to your external. Help messages may reassure them. But only experience teaches them. If your external permanently modifies the drawing in any way, you must preserve the user's guarantee to experiment freely by providing undo. VectorWorks offers a simple, powerful and generic undo facility for use in an external. For information on how to implement undo, see the subsection on undo in the services section.

• Balloon help - All externals have balloon help that concisely states the purpose of the external. It appears when the user positions the cursor over the tool or menu command.

• Message bar help - When a tool is selected text can be displayed in the message bar. An external should display the name of the current tool mode, and prompt the user towards the next expected action

• Smart Cursor - VectorWorks uses the cursor to provide information to the user about a tool. Which cursor a tool uses should indicate to the user what task the tool will perform next. An example is how the selection tool changes from a hollow arrow when the cursor is over an object to the standard arrow cursor when the next click will allow specification of a selection marquee. Make use of existing VectorWorks cursors and conventions where possible and make logical extensions otherwise. Some tools may need a completely new cursor. If a tool's behavior depends on the object it is being used with or on some state internal to the tool, then the appearance of the cursor should convey this. Tutorial #2 implements a tool that makes good use of the smart cursor mechanism.

• Alerts - If the user attempts to use the external in a way that is inappropriate or impossible, the external should provide information about what is wrong and how to correct the situation. An external should never fail without an explanation or with only a beep.

• Constraints - Constraints allow the user to control an external's behavior and enhance its usefulness. See the constraints subsection in the services section for a discussion of how to make a tool work with the constraint palette.

• Menu Highlighting - Menu commands have the option to make themselves unavailable when the conditions for their use are not met. Disabling a menu, however, does not provide a user information, it only limits their choices. A menu should not disable itself unless it is clear to the user why the functionality is inappropriate at that time. If it is unclear then the menu command should remain enabled. If the user selects it, it can provide an alert telling the user it is unavailable. The alert should explain what conditions need to change for the menu command to be activated.

The Definition of an External

Implementation of an external can be broken into two parts: writing the program code that defines the external and creating the proper non-code resources. These components are combined into a single file, only one external per file, that becomes the document that the user sees and places into their "Plug-ins" folder. The following resources will be found in a completed external:

• Code resources. This resource contains the compiled code of the external, and should always have a type of XCDD and an ID of 100. The code for any external contains a main function with parameters defined for the external type, and any number of support functions. The main function is the entry point routine that gets called when VectorWorks calls the external. Typically the main routine is specified by the name 'main', but some development systems allow you to specify an arbitrarily named function as the entry point. Another code resource of type XCOD and id 50 will also be appended to the external.

• Definition Resource. Each type of external has a resource type that defines certain standard behavior for the external. For a menu command the resource type is MITM, for a tool the type is TDef. Every external must have a valid definition resource with an ID of 128.

• Auxiliary Resources. Required or suggested resources such as text for balloon help and additional cursors for tools.

TEXT
This resource holds the text for the externals balloon help and must be numbered with resource ID 128. The length of this resource must be less than 239 characters. Balloon help is managed by VectorWorks automatically when you provide this resource. If you add or edit this resource in the external you will need to edit all Workspaces containing that external to force the new balloon help to appear.

vers
All externals must have vers resources of ID 1 and optionally of ID 2. 'vers' 1 is the external's version. 'vers' 2 is the package version that the external was shipped with. Diehl Graphsoft externals use 'vers' 2 to specify which version of VectorWorks the external shipped with. If your external exists independently then it should not have a 'vers' 2 resource.

• Private Resources. Externals may also use their own resources, as detailed in Inside Macintosh. So as not to conflict with resources in VectorWorks, resource ID for private resources should be in the following ranges:

PICT, ICON, ICN#, DLOG, DITL, STR#, ALRT, FONT, dctb, DLGX, MMAP: 11000 to 14999

CURS: greater than or equal to 17000
MENU: 128-191 for hierarchical menus, otherwise greater than or equal to 10000
All other resource types: greater than or equal to 10000

Tool or Menu Command?

Many Externals could possibly be written as tools or menu commands, but there is usually a compelling reason to choose one over the other. Functionality that should have been implemented as a menu command that is wrapped in a tool may seem awkward to a user and vice versa. If the user doesn't feel comfortable with your external, then no matter how important you may think it is to them, they probably will not use it. When deciding how to implement your external, consider the following guidelines:

• Menu Commands: Menu commands usually interact with the user only through dialogs and often don't have any interface. Menu commands are appropriate when an action is to be performed on the selection or the document as a whole.

• Tools: A tool is the appropriate choice if the user will need to interact with the drawing or if mouse input is required. Unless you are implementing a new selection tool, the primary focus of a tool's interaction should not be selecting the objects to operate on. If you find that this is the case, try to implement your external as a menu command and let the user select the objects with the selection tool. Good examples of tools are creation tools and the 3D view tools.

Menu Commands

The SDK supports several types of Menu Commands. An external Menu Command can be either a single basic menu command or a block of several related menu commands, which is referred to as a menu chunk.

A Basic menu command implements a single menu item in a menu. Basic menu commands have no support for item checking, run-time item disabling, or multiple commands in a single external. If you need any of these behaviors you can create a menu chunk.

A menu chunk displays several menu commands together in a menu, and shares the same source code. This is often desirable when the user needs to select between a group of related options that perform a very similar function.

All menus in VectorWorks, both internal and external, use the same framework of MITM resources, MMNU resources, and menu definition functions. Chunks, being the most general type of menu items, must support the widest array of options. Because of their generality, chunks require more setup and have more associated complications than single menu items.

There are three types of chunks:

Plain chunks contain a fixed number of items that are defined in the resource file in an MMNU resource. The size of these is fixed when they are implemented.

Dynamic chunks are constructed at runtime by the menu definition function. They can be any size, but the size cannot be changed after initialization. The Workspaces menu is an example of a dynamic chunk.

Variable-dynamic chunks are similar to dynamic chunks in that they are constructed at runtime. They are more flexible, however, because they can change their size at any time. The Layers menu, and Windows menu are examples of variable-dynamic chunks.

Because dynamic and variable-dynamic chunks have an unknown size when the menus are being built, the Workspace Editor only allows them to be placed at the end of a menu. Typically, they are placed in a hierarchical menu.

Menu Command Main Function

The prototype for a Menu Command's main function is:

```
extern "C" long main(long action, long message, long& userData,
CallBackPtr cbp)
```

The parameters to the main function are:

`action` Identifies the task the menu command needs to perform, and determines the significance of the message parameter. The actions are described in detail below.

`message` This parameter's meaning depends on the value of the action parameter and the type of menu command. For some actions, it is unused. For chunks, the message parameter will often be a pointer to the following structure as defined in MiniCadHookIntf.h:

```
struct MenuChunkInfo {
      short menuID;
      short itemID;
      short modifiers;
      union {
            short chunkIndex;     // for kMenuDoInterface
            short chunkSize;      // for kMenuCheckHilite
}
```

The meaning of these fields varies slightly for different actions and is explained below.

`userData` Four bytes of data that VectorWorks maintains for the external between calls. This is necessary because an external cannot directly maintain its state between calls. The section on the Development Environment explains in great detail the peculiarities of externals and global variables.

`cbp` An implementation detail. See the documentation in Section III for more information.

Menu Command Actions

The menu command main function will be called by VectorWorks, and will be passed one of several values for the action parameter indicating the task to be performed. Basic and Chunk menu commands must respond to the following values for the action parameter:

`kMenuInitGlobals`

A workspace containing the external has been loaded. This is the external's opportunity to allocate and initialize all persistent data. A Macintosh handle to the data should be assigned to the userData parameter so that it can be used later for other actions. The message parameter is unused. This action should always return 0. This action will not be called again until a kMenuDisposeGlobals is passed.

`kMenuDisposeGlobals`

A workspace containing the external has been deselected. The menu command should finalize all outstanding actions and free any allocated data and resources. The message parameter is unused. This action should always return 0.

`kMenuDoInterface`

The user has selected the menu command. The menu command should take whatever action is appropriate to implement its functionality. For basic menu commands the message parameter is

unused. For chunks, the message parameter is a pointer to a MenuChunkInfo structure with the following fields:

```
mc->menuID; // Menu id of the menu containing this chunk
mc->itemID; // Item number of item before chunk's first item
mc->chunkIndex;    // Index of the chunk item selected
                   // where 1 is first chunk item.
mc->modifiers;     // Modifiers from the mousedown event
```

Chunks should avoid using the modifier field because it may prevent the chunk from being called by a VectorScript.

The menu command should return kMenuDidNotChangeDoc if it did not change the document (this allows VectorWorks to activate and deactivate the save command when appropriate).

Chunks receive the following additional action values:

`kMenuItemEnabled`

This action occurs when the menu item is called from a VectorScript using the DoMenuTextByName procedure. This action allows the system to efficiently determine whether a given menu item is enabled in order to return appropriate errors when the menu function is called from VectorScript or some other script system.

The message parameter is 0 if this is a basic menu item or the chunk index of the desired item if it's a chunk. You should examine the message parameter, efficiently determine whether this item is enabled or disabled, and return kMenuItemEnabled or kMenuItemDisabled. If the chunk index is not legal, (for example if it's larger than the number of items in your chunk), return kMenuNoSuchItem.

`kMenuCheckHilite`

When the user clicks in the menu bar, this action is sent to all chunks in the menu system to allow them to prepare for display to the user. You can set the enabled state, set check marks, and change the text style for any or all of the items in your chunk. If your chunk is variable, you can delete items or add new items in response to the current state of the drawing. You can use any standard menu manager routines to set up your chunk for display to the user.

Since all this must happen after the user clicks the mouse but before the menu is displayed, this is a time critical section of code. You should make every effort to avoid doing unnecessary work when you get this action.

The message parameter is a pointer to a MenuChunkInfo structure with the following fields:

```
mc->menuID; // The menu id of the menu containing this chunk
mc->itemID; // The itemID of the first item in the chunk
mc->chunkSize;     // The number of items in this chunk
mc->modifiers;     // The modifiers from the mousedown event
```

Return true if this call is necessary, false otherwise. (Return true if any items were disabled.)

Note: Avoid using the modifiers field of the MenuChunkInfo. Your menu item cannot be called correctly from VectorScript or other script systems if it depends on this information.

Warning: The resource fork of your external file is not open when your code gets this message (for speed reasons). Your code cannot load resources from its file or make calls to code outside the main segment. This is a severe restriction, so be very careful when implementing the code for this action.

`kMenuNotify`

This action notifies the menu chunk that a particular VectorWorks internal event has occurred. The menu must have previously registered for the notification. This is typically done in the kMenuInitGlobals action. For example, to register for the "Document Changed" notification you would call:

```
RegisterMenuForCallback(cbp,GetMyMenuCommandIndex(cbp),'DOCC');
```

The menu will continue to receive a kMenuNotify action every time the active document changes until it unregisters by calling the function:

```
UnregisterMenuForCallback(cbp,GetMyMenuCommandIndex(cbp),'DOCC');
```

The message parameter for this action indicates which event has occurred. The message is a 32-bit value of type OSType that can be represented as a four-character code. For example, the menu can test the message value as follows:

```
If (message == 'DOCC') {
      (*(MyInfoHandle)userData)->needsDocRebuild = true;
}
```

The menu can respond to the notification immediately or it can set a global flag for use later (as shown in the example above). Often it is best to postpone any significant work until it is required.

This action should always return 0.

Warning: The resource fork of your external file is not open when your code gets this message (for speed reasons). Your code cannot load resources from its file or make calls to code outside the main segment. This is a severe restriction, so be very careful when implementing the code for this action.

`kMenuAddItems`

The menu that contains this chunk has been constructed up to the item before your chunk. Add any items you want in your chunk to the end of this menu.

This action is only passed to dynamic and variable-dynamic chunks. Basic menu commands and plain chunks do not receive this action. It occurs once, immediately after kMenuInitGlobals and before any other action. If a variable-dynamic chunk can determine all of its items at this point it should do so.

The message parameter is a pointer to a MenuChunkInfo structure with the fields:

```
mc->menuID; // Menu id of the menu containing this chunk
mc->itemID; // Item number of item before the chunk's first item
mc->chunkIndex;    // unused
mc->modifiers;     // unused
```

Typecast the message parameter and then use it to retrieve the MenuHandle to your menu. Using this handle, you can add your chunk's items to the end of the menu with AppendMenu as follows:

```
MenuChunkInfoPtr mc = (MenuChunkInfoPtr) message;
MenuHandle myMenu = GetMenuHandle(mc->menuID);
AppendMenu(myMenu, "\pItem One");
AppendMenu(myMenu, "\pItem Two");
AppendMenu(myMenu, "\pItem Three");
```

If you would like to display the special characters ; ! > / (in your menu item, you must use the following technique:

```
AppendMenu(myMenu, "\pdummy");
SetItem(myMenu, mc->itemID+1, "\pCalculate miles / gallon");
```

This action should always return 0.

If the items in your variable-dynamic chunk can change during the execution of VectorWorks then the MenuHandle can be updated during the kMenuCheckHilite action.

Note: Very few of Diehl Graphsoft's menu commands are implemented as chunks, as the basic behavior is almost always sufficient.

Menu Command Definition Resource: MITM

All external menus are defined by their MITM 128 resource, which is contained in the external menu file. This resource is an extension of the item format used in MMNU resources so they don't have to be parsed separately. As such, the MITM structure contains some data that is unused. The fields described below are used by chunks. All other fields should be 0.

The fields of the MITM resource are defined as follows:

Needs and NeedsNot

These two fields each contain one word of flags, which determine the default highlighting behavior of the menu item. Needs specifies conditions which must exist for the menu item to be available, and NeedsNot specifies conditions which must not exist. The meaning of each bit of the two flag fields is identical:

Bit	Meaning
0	VectorWorks Document is active.
1	Reserved
2	Reserved
3	Reserved
4	Reserved
5	Reserved
6	Reserved
7	There are one or more objects selected.
8	There is more than one object selected.
9	The selection contains at least one 2D object.
10	The selection contains at least one 3D object.
11	The current layer is in plan projection.
12	Reserved
13	Reserved
14	Reserved
15	Reserved

If Needs = 0 and NeedsNot = 0 then the item is always active. If Needs and NeedsNot both have a corresponding flag set, the item is always inactive. The reserved flags should not be set and in particular Needs = 0xFFFF and NeedsNot = 0xFFFF is a reserved setting.

You may wish to inspect the flag fields for VectorWorks' internal menu items by using ResEdit and inspecting the MITM resources in the VectorWorks application.

For menu chunks, the Needs and NeedsNot flags will enable and disable every item in the chunk. Before the selected menu bar is displayed to the user, kMenuCheckHilight actions are sent to all chunks so that they can modify the checked or hilighted state of each of their items.

Title

This string determines which name the user will see when in VectorWorks or Workspace Editor, and is also the string that can be changed to localize the external for other languages. The actual file name of the external should never be changed, however, because this is how VectorWorks finds the external.

commandID

These are private fields used by VectorWorks and should always be 0.

Chunk

This flag indicates that this menu function implements a menu chunk.

Has Option Equivalent

Unused flag that should be set to False or 0.

Dynamic

This indicates that the chunk is built on the fly rather than being stored in an MMNU resource. The Dynamic flag only applies to chunks.

Variable

This flag indicates that the chunk can change size while VectorWorks is running. This flag only applies to dynamic chunks. Variable chunks can be placed only at the end of a menu in the Workspace Editor.

Reserved

These four flags are reserved for future use and should be set to False or 0.

iconID, keyEquivalent, keyFlags, mark Char/ sub menu id, style flags

The MITM resource describes a data structure that is a superset of Apple's item definition contained in the variable length data section of a MENU resource (Inside Macintosh: Toolbox Essentials 3-154). As such, the MITM fields with the same name as fields in the MENU usually have the same meaning. However, in the case of icons, there is a subtle difference. Inside Macintosh states that the iconID should equal the icon's resource ID -256. In an MITM, though, the iconID should equal the icon's ID.

For plain chunks (not dynamic), the "mark Char/ sub menu id" field of the MITM must contain the resource id of an MMNU resource which defines the titles of the chunk's menu items. For dynamic chunks this field should be 0.

chunk, dynamic, variable

To specify that your menu command accepts one of the extended message sets, set the appropriate flags to true. If your menu command is a chunk, but not variable or dynamic, then you will also need to provide a MMNU resource that describes the items in your chunk. Also, you will need to set the mark Char field to the ID of your MMNU.

Tools

Tool Main Function

The prototype for a Tool's main function is:

```
extern "C" long main(long action, long message1, long message2, long&
userData, CallBackPtr cbp)
```

The parameters to the main function are:

`action` Identifies the task the tool needs to perform, and determines the significance of the message parameters.

`message1` The meaning of this parameter depends on the action.

`message2` The meaning of this parameter depends on the action.

`userData` Four bytes of data that VectorWorks maintains for the external between calls. This is necessary because an external cannot directly maintain its state between calls. The section on the Development Environment explains in great detail the peculiarities of externals and global variables.

`cbp` An implementation detail. See the documentation in Section III for more information.

Tool Actions

Tools receive the following action requests:

`kToolInitGlobals`

A workspace containing the external has been loaded. This action will not be called again until kToolDisposeGlobals has been called. This is the external's opportunity to initialize all persistent data. The message parameters are unused. This action should return 0 if initialization succeeded, otherwise return kToolCouldNotInitialize.

`kToolDisposeGlobals`

A workspace containing the external has been deselected. The tool should finalize all outstanding actions and free any allocated resources. The message parameters are unused. This routine should always return 0.

`kToolDoSetup`

The user has just selected the tool. At this point you should set the help string and if the tool uses modes, setup the mode bar. If you make use of custom data fields, set them up also. The message parameters are unused. This action should return 0 if initialization succeeded, otherwise return kToolCouldNotSetUp.

`kToolDoSetDown`

The user has selected another tool. The message parameters are unused. This action should always return 0.

`kToolDoPick`

Before the user clicks on the document with a tool, VectorWorks indicates the upcoming action by changing the cursor to reflect the mouse position on screen. The message parameters are unused. This routine should return the ID of the CURS resource that should be displayed. Returning 0 will cause the cursor specified in the tool definition resource to be displayed.

`kToolDoInterface`

The user has clicked in the document and the external should begin interacting with the user. message1 is a pointer to a CoordPt containing the position of the mouse in world coordinates. message2 is a Point set to the position of the mouse on the screen. Although the screen coordinates of the mouse are provided, there are very few reasons to use them as all drawing and calculations should be done in world-space. If the tool did not change the document it should return kToolDidNotChangeDoc. Additionally, if the tool can no longer be the active tool it should return kToolSwitchToCursor.

`kToolDoModeEvent`

The user has selected a mode in the mode bar. message1 is the index of the group the user clicked in. The high word of message2 is the old selection, the low word is the new selection. This action should always return 0.

`kToolDoDoubleClick`

The user has double-clicked on the tool's icon. If the tool is a creation tool then it should provide the user with a dialog to specify the new object. This is currently the only behavior implemented in VectorWorks in response to this action. Expanding the meaning of this action should be done with care. In particular, do not use it as an alternative interface to behavior for which VectorWorks provides a standard for access (for example: do not bring up a preference dialog in response to this action. Preferences should be accessed through a mode bar button).

Tool Definition Resource: TDef

Every tool has a TDef resource of ID 128. Each of the fields of the TDef resource are described below:

disablePickUpdate

When set to true, the automatic updating of the Tool Pick Record is disabled. Normally, as the mouse is moved over the document, VectorWorks automatically finds out what objects are near the mouse for the kToolDoPick action. This may take a significant amount of time on large documents. Tools which do not need this functionality, such as pure creation tools, should set this flag.

needsPlan

If a tool requires the current projection to be set to plan, set this flag.

needs3DProjection

If a tool requires a non-plan projection, set this flag.

uses3DCursor

Tools that operate using 3D grid and snapping points should set this flag.

screenBasedCursor

Activating this flag disables all gridding and snapping of the mouse when functions such as GetCoords are called.

dontShowScreenHints

Tools for which the smart mouse screen hints are not useful can turn off the display of those messages by setting this flag.

iconID

The iconID field indicates the resource ID of the icon to be displayed in the tool palette, which should also be included in the resource file.

defaultCursID;

If the Tool definition function returns 0 in response to a kToolDoPick message, then VectorWorks sets the cursor to the one specified by defaultCursID.

messageStr

When the tool is selected (before the kToolDoSetup message has been passed to the definition function) this string is set into the message bar.

waitMoveDist

For many tools, a good user interface allows a bit of "slop" before the tool actually takes effect. The value of waitMoveDist is a distance in screen pixels that the mouse must move before VectorWorks passes the kToolDoInterface action code to the tool definition function. If this distance is not exceeded before the tool click is finished, then a selection tool is automatically made active.

constraintFlags

This bit-field determines which cosntraints the user has available when the tool is active. Each bit represents one of the constraints, with 1 indicating an available constraint. For the 10 current constraints, adding together a combination of the following values will enable the appropriate constraints:

1	Snap to Grid
2	Snap to Objects
4	Snap to Surface/Working Plane
8	Snap to Intersection
16	Snap to Distance
32	Constrain Parallel
64	Constrain Perpendicular
128	Constrain to Angle
256	Constrain Symmetrical
512	Constrain Tangent/to Working Plane Normal

For example, to activate only the Snap to Grid and the Snap to Intersection constraints, set constraintFlags = 1+8 = 9.

VectorWorks tools generally fall into one of three "families" of constraint availability depending upon which type of geometry the user is setting: points (locus tool), lines (segment tool, polygon tool), or boxes (ellipse by rectangle tool and rectangle tool). One of these three sets of constraints flags often contains the correct constraintFlags for new tools.

Min Version

This field holds the earliest version of VectorWorks that will properly run an external. Each callback contains the version of VectorWorks that first supported it, put the latest callback version in the Min Version field.

Databar Mode

When the user clicks into the document and before the tool external receives a kToolDoInterface message, VectorWorks sets the Data Display Bar to the mode specified here. The file `MiniCadCallbacks.h` lists the available constants. If the external does not need to change the Data Display Bar or needs to set a custom set of bars, set this field to 0.

Tool Icon

All Tool externals must include an ICON resource. This icon is the icon that appears on the tool palette in VectorWorks. The ID must be greater than 11000 and must match the corresponding field in the TDef resource.

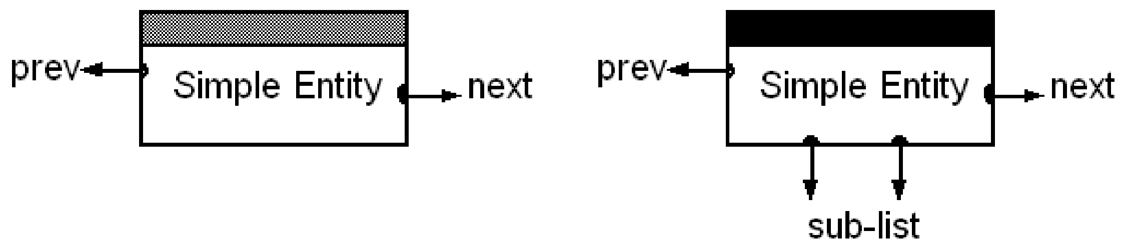
Section II: The VectorWorks Environment

Basic Data Types

See the file `MCCoordTypes.h` for the formal definition of the following types:

Entities

The simplest element of the VectorWorks environment is the entity. Every entity has a type (found by calling `GetObjectType`) which indicates what other properties it has, a position in a drawing list (found by calling `NextObject`, `PrevObject`, and `Parent`), and an auxiliary list (described below). In addition, some entities, called containers, also hold a sub-list of entities and can return the first and last entities in that sub-list. Layers and Groups are examples of container entities.



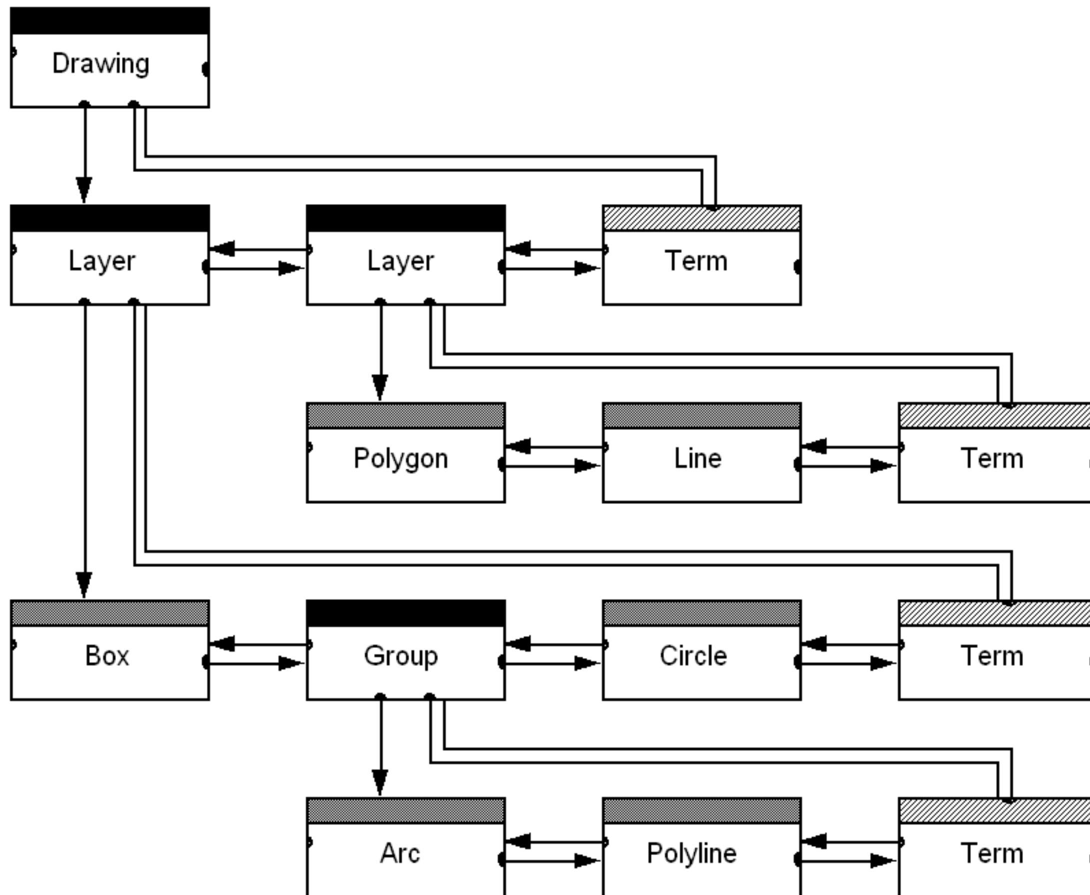
Lists

VectorWorks uses two types of lists to hold drawing data, explicitly and non-explicitly terminated lists. Explicitly terminated lists maintain a node at the very end of the list of the type termNode. This node is used as both a signal of the end of the list and more importantly to store a reference back up the tree to the list's parent. Non-explicitly terminated lists do not have a special termination node, and the end of the list is indicated by a nil reference in the last node's next pointer. Because this type of list does not have a terminal node, it is not possible to backtrack up the tree to the parent without maintaining a reference to the parent yourself. All lists of drawing objects are explicitly terminated.

In addition to drawing object lists, there is another type of list, the auxiliary list. Aux lists provide for supplementary data to be associated with an object (such as record data or cavity lines for a wall) that is saved with the document. An aux list is an unsorted, non-explicitly terminated list that can be searched by keys. Some records in an object's aux list may have public interfaces, but most data in the aux lists is private and should not be used.

VectorWorks defines a special type of aux object for use by external writers, the userDataNode. It is an aux object with a type of userDataNode and an internally maintained tag of type `OSType` that the external writer can use as a signature. To create a userDataNode, define a class or record that descends from UserDataType (defined in `MCCallBacks.h`) or which contains a UserDataType object as the first member. Either method will reserve the necessary space at the head of the structure for VectorWorks use. Next, call `NewDataObject`, passing it the size and signature of your private DataNodeType. You now have an aux object that can be safely used with the supplied aux lists callbacks. For convenience `FindDataNode` has been supplied to simplify locating your private data objects.

Drawings



VectorWorks Document Structure and List Management

The drawing list can be viewed as a tree. Its root is called the drawing header. The drawing header is a container entity whose sub-list contains all the entities in the drawing. The first level of sub-objects in the drawing header is layers. The drawing header is the only object that can contain layers, and it contains every layer in the drawing. A layer contains all the entities that are on that layer, but not every entity on the layer is in the layers sub-list. Some entities on the layer are contained within other entities on the layer. A rectangle in a group on a layer is contained by both the group and the layer but only appears in the sub-list of the group. Since the group is in the layer's sub-list, however, the layer does indirectly contain the rectangle.

The auxiliary list of the drawing header holds all the defining entities global to the document. These include symbol definitions, record formats, worksheets, and command palettes.

List Traversal

Traversal is useful because it allows you to perform an operation on a set of drawing objects that meet some criteria that you specify. For example you might want to:

- Move all selected objects three inches to the left,

- Resize all selected objects that are arcs of angles less than ten degrees to ten degree arcs,
- Find every wall in the current layer and insert studs into the layer '<current> construction'

VectorWorks provides two methods to search the drawing list. The first and preferred method is through the routine `ForEachObject`. `ForEachObject` can perform either a shallow or deep traversal, in drawing order (back to front) starting from any point in the tree. If you use it to do a deep traversal from the drawing header it will iterate over every object in the drawing. If, instead, you did a shallow traversal from the drawing header it would only return references to the layer objects in the drawing. `ForEachObject` also takes as a parameter a selection clause that allows you to specify certain constraints that an object must meet to be enumerated. This filtering is limited to the following attributes: is the object selected, editable, a drawing object, or a symbol definition. The last criterion is a special case as symbol definitions are not in the drawing tree, only symbol instances are. Symbol definitions are kept in a separate tree also rooted at the drawing header. They are organized according to the same method drawing objects are (as symbols are comprised of drawing objects). More information on the symbol definition list and symbol definitions can be found in the section on Special Objects. Filtering based on other criteria must be hand coded.

`ForEachObject` works by calling a user supplied function once for every object it finds. The user-supplied function must be of the type `ForEachObjectProcPtr`.

```
typedef void (*ForEachObjectProcPtr)(Handle h, CallBackPtr cbp, void
*env);
```

The argument "h" is a handle to the object `ForEachObject` found. cbp is a parameter used in communicating between an external and VectorWorks; it is discussed in detail in the section on the Development Environment. env is a pointer to data supplied as a parameter to `ForEachObject`. It is through env that the calling function can share data with the called function. See the tutorials for examples of `ForEachObject`. It is one of the most important and frequently used functions VectorWorks provides. It is also one of the more difficult to use correctly, so take time to study the examples.

The second method for traversing the drawing tree is through standard tree walking functions. VectorWorks provides routines to move between siblings in the tree, `NextObject` and `PrevObject`; to move to the parent of a node, `ParentObject`; and to move to the children of a node, `FirstMemberObject` and `LastMemberObject`. Using these functions you can implement any type of tree walking algorithm. This flexibility is more difficult to implement and manage than `ForEachObject` so it should only be used when necessary.

Graphic Entities

Units

Before discussing some specific information about entities, an understanding of the different coordinate spaces used by VectorWorks is important. VectorWorks defines and manages the space through coordinate systems it imposes on it. The following are the basic coordinate systems:

Model

> The units presented to the user. VectorWorks never expects these units as parameters and uses them only for display of information to the user.

World

> This is the VectorWorks universe. At run-time it's represented by the white space on screen, and its border by the gray that surrounds it. It is finite, discrete, and dependent upon the units of the drawing. Every point can be represented by a 32-bit long integer quantity of the VectorWorks defined type `Coord`. The precision of a `Coord` value depends on the scale and paper size of the drawing. A coordinate in model space = (world space) / (stored accuracy) + (user origin offset). The stored accuracy is defined in the UnitsType structure that is described below.

Local

Some VectorWorks entities (symbol definitions, extrudes, multiple extrudes, sweeps, and layer links) use simple entities to define their geometry and store these defining entities in local coordinates. This coordinate system differs from World space only in the offset used to convert to model space.

View

This coordinate system is discussed only for completeness, and few situations require use of this coordinate space. A 2D-only space which represents the viewable area of the document window, this coordinate system is used only for drawing to the screen. The mapping from world space to view space changes frequently so view space coordinates are usually useless for anything but immediate drawing. If you wish to draw something to the screen, use the World space drawing routines provided. The basic type of this system is the QuickDraw Point, which represents space in pixels. As the coordinates in QuickDraw space are 16-bit values there is not a unique mapping between world and screen space (many World space points map to the same View space point).

Page

This is a 2D-only coordinate system that represents the physical page the drawing is on. It basic type is the `double_t` and it's unit is always inches. It is unaffected by any changes to scale or units.

General Properties of Drawing Entities

In the last section, entities were discussed in relation to the drawing list. Drawing entities also have some other common properties other than their position in the drawing hierarchy.

Graphic Attributes

Entities have the following attributes for which VectorWorks provides simple accessor and mutator functions: name, class, lock state, color, fill pattern, pen pattern, line weight, visibility, and arrow heads. VectorWorks also provides routines to get and set the default values of these attributes.

Defining Geometry

All entities have a location in World space, but the method for storing the defining geometry differs for each entity. You can determine an entity's location by examining its bounding rectangle using `GetBox` (which returns the smallest rectangle that completely surrounds the entity in the current view) or `GetCube` (which returns the smallest cube that completely surrounds the entity). For many entities (such as walls and groups), this rectangle or cube is a derived attribute, an attribute constructed from the actual defining geometry of the entity. This distinction has subtle but important consequences since some functions such as `SetBox` only have lasting effects on entities with defining rectangles (such as rectangle and ellipses).

Special Objects

VectorWorks has entities of many different levels of complexity. These range from rectangles and line segments, which have trivial defining properties to symbols and extrudes which hold information in local coordinates systems which get transformed to their final location. This section explains some of the more complicated entity types.

Symbols

There are two drawing entities related to symbols, symbol instances and symbol definitions. Symbol definitions contain the entities that define the symbol and are stored in the aux list of the drawing header. Symbol instances are markers that are placed in the drawing list specifying a transformation matrix and a symbol definition to draw there.

Symbol Definitions

A symbol definition is very much like a VectorWorks group. The two distinguishing aspects are the local coordinate system and their location in the drawing. Why do symbol definitions need a local coordinate system? Groups do not need to use a local coordinate system because every instance of a group is unique.

You can copy and duplicate a group but that act creates a new and distinct instance of the group. Changes to the new group do not affect the old group and vice versa. Because this is true, when you move a group in the drawing you can directly change the coordinates of the component objects. Symbols, however, are a shared object. If you change any component object in a symbol, all symbol instances will reflect that change. To allow for the same symbol definition to be displayed at multiple locations, a local coordinate system is used. In this method, all the component objects in a symbol are specified relative to the symbol's coordinate origin. When a symbol instance is made, the symbol's component entities are drawn as if the insertion point were the symbol's local origin.

If you try to create a symbol from a group of objects on the drawing, that symbol's insertion point will be the World space origin. This will probably be the wrong effect if the collection of entities does not lie near the World space origin. To deal with this, choose some World space point to be the symbol's insertion point. Then for each object you want to add to the symbol call `MoveObject3D`(myInsertionPoint). If you are creating the objects in the symbol from scratch, you can create them about any arbitrary point and then proceed as above or you can create them about the world origin.

Symbol Instances
Symbol Instances are fairly simple objects. They contain a transformation matrix and a symbol definition reference. To draw the symbol instance, VectorWorks takes the geometry in the symbol definition and multiplies it by the symbol instance's matrix. Modification of the instance's symbol definition is done by calling `GetDefinition` and `SetDefinition`. VectorWorks also provides two creation routines for adding new symbol instances to the drawing: `PlaceSymbol` and `PlaceSymbolByName`.

2D and 3D Symbols
Symbols have the additional complication of being hybrid entities, entities that can have both 2D and 3D properties. VectorWorks stores both "halves" of the symbol in the same definition list, and then filters out entities as it processes the symbol for different operations. `GetSymbolType` can be called to quickly check the type of a symbol.

The Symbol Library
Since symbol definitions do not directly appear in the drawing, they are not stored in the main portion of the drawing list. Instead they are stored in their own list, which is found in the drawing header's auxiliary list. To get access to the symbol library you can call `GetSymbolLibraryHeader` and then traverse the symbol library with the routines discussed in the drawing list section or you can use `ForEachObject` specifying that you only want symbol definitions.

Walls
Walls are defined as a line segment with a variable number of "nodes", called breaks, that lie along its length. The controlling line, accessed with `GetEndPoints`, specifies the center line of the wall. The width of the wall (`GetWallWidth`), indicates how far on each side of the center line to create the outer edge of the wall. For 3D drawing, heights of the start and end of the wall (`GetWallHeights`) can be independently set.

Breaks are modifications to the wall. An offset from the start point of the wall specifies their location. Their type can be one of the following:

Cap

> A Cap break specifies modifications to an end of the wall. A wall may have 0, 1 or 2 caps. The offset of a cap break should be -1 if it is the start cap and LONG_MAX (defined in limits.h) if it is the end cap. A cap specifies whether the end of the wall is closed and if it is closed whether it is rounded. Since the position of the wall is specified by the center line, the cap also specifies any offset of the endpoint of either side line from the corresponding end point of the center line (to create a "tilted cap"). VectorWorks ignores cap end offsets with round caps.

Half

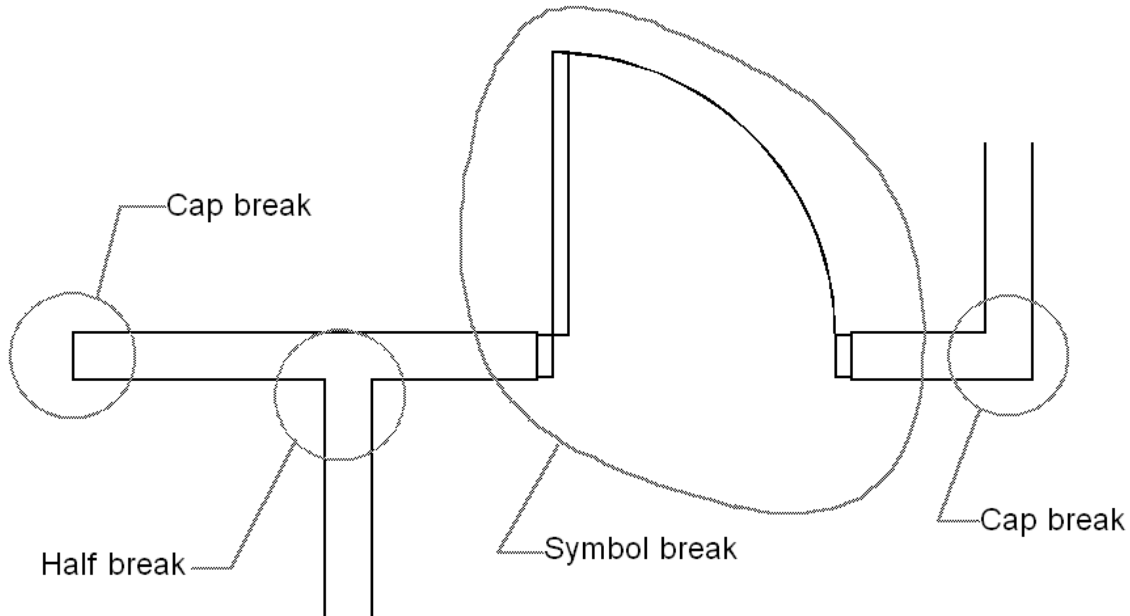
A half break is a gap in a side of a wall visible in plan view. These occur where one wall joins another. A half break defines a starting position for the gap and a gap length.

Peak

A peak break is a 3D only break that specifies the height of the wall at a given offset.

Symbol

A symbol break specifies the insertion of an instance of a symbol into a wall. The symbol cuts the wall as necessary. A symbol break defines the symbol definition of the inserted symbol, the height of the symbol's insertion point in the wall, and the orientation of the symbol (towards which quadrant of the wall is the symbol directed).



Extrudes

An extrude is a container entity and may be considered single or multiple. The extrude also has a top and bottom offset value (GetExtrudeValues) to specify the top and bottom surfaces of the extrude, and a transformation matrix. To convert the defining geometry from 2D local coordinates to the 3D World space points, simply take the 2D coordinate, take the bottom surface value (to find the corresponding bottom point in 3D) as the z-coordinate of that point and multiply it by the extrude's matrix.

Multiple extrudes use the same values as single extrudes, but distribute each entity equally between the top and bottom surfaces, in back to front list order.

Sweeps

Sweeps, like extrudes, are container entities that have a sub-list of 2D defining entities and a transformation matrix. To convert the entities into the 3D swept shape, each 2D point is rotated around the y-axis in the local coordinate system, and the resulting 3D point is then multiplied by the sweep's transformation to orient it correctly.

Meshes

Meshes have a complicated internal storage structure to optimize them for both speed and size. For editing purposes they are converted into and from a group of 3D polygons (MeshToGroup and GroupToMesh). When converting from 3D polygon groups, shared edges of polygons become single mesh lines in the mesh. This is the only specific editing method for meshes.

Services

3D Views
An external can specify a camera location and orientation with a TransformMatrix. The matrix can be constructed and manipulated with the vector and matrix routines in the MCVector and MCMatrix libraries. The matrix should transform 3D objects from world space to view space, which is then projected onto the page according to the current projection. The projection plane (i.e. the screen) is defined to be the view space x-y plane and the positive z extends toward the viewer (i.e. out of the screen).

Additional assumptions are made when the current projection is perspective. The viewer location is considered to be the point (0, 0, perspectiveDistance) in view space. Objects are clipped to the view volume which is that space extending from the viewer location toward the projection plane and enclosed by the clipRect on that plane. Finally, all objects are clipped to the back clipping plane which is specified by z = clipDistance in view space. The values of clipDistance, perspectiveDistance, and clipRect are obtained from the GetPerspectiveInfo callback.

There is also a set of routines supporting interactive rotation of 3D objects. This is the behavior the user sees when using 3D view specification tools like walkthrough and flyover. To use this service, first get the current view matrix with GetViewMatrix. Then create a preview with PreviewSetup. You can then repeatedly modify the matrix, call SetViewMatrix and call PreviewDrawFrame in a loop until the desired view is chosen. The view change would typically respond to input from the user, but QuickTime™ export uses the same calls to display animation previews. You must then call PreviewFinish, and if you want to permanently alter the view, you must call NewCurrentViewMatrix to set the new view.

Using Undo
VectorWorks provides a general method for undo that is sufficient for most purposes, though it can be wasteful of memory. This mechanism for undo works by maintaining three lists of objects. The first list contains copies of the all objects deleted by an operation. If your external deletes an object call AddBeforeSwapObject before you delete it. The second list contains a copy and a reference to all objects modified by an operation. Before the external modifies an object it should call AddBothSwapObject. The third list contains references to all objects created by an operation. Call AddAfterSwapObject for every object your external creates.

When the user selects undo, any object in the “before” list is reinserted into the drawing, any object in the “after” list is deleted from the drawing, and any object in the “both” list is removed and replaced by the copy of the original object. If the user chooses to redo the operation everything is redone correctly.

Display of Numeric Values
When displaying numeric data to the user, it is important that the format of the numbers conform to the user’s specifications, and where those are incomplete that they be consistent. For most needs, the VectorWorks routines NumToDimString and AngleToString will suffice. If you are working with non-coordinate values, or floating point coordinate values, you may need to format the numbers yourself. To do this you will need access to the users specifications for number display. This information is stored in a UnitsType record (see `MiniCadCallbacks.h`).

```
struct UnitsType {
      Coord             storedAccuracy;
      short             format;
      Extended80        unitsPerInch;
      Str7              unitMark;
      long              displayAccuracy;
      UnitFlagsType     unitFlags;
};
```

Note: World coordinates are centered on the world origin, which never moves. If the user has moved the user origin, then you will need to offset world values by the user origin's location in world space before you display them. To get the user origin use GetUserOrigin.

Section III: The Development Environment

Cross-Platform Development

The VectorWorks SDK enables development of externals for both the Apple Macintosh and the Microsoft Windows operating systems. A separate development environment is used on each platform to build an external for that particular platform. Depending on the specifics of the external, some of the source code may be shared between the two versions and some may be platform-specific.

On the Macintosh, the supported development environment is Metrowerks CodeWarrior Pro 4, and on Windows it is Microsoft Developer Studio Visual C++ 6.0.

This section of the manual discusses the development environment on the Apple Macintosh platform.

Project and Compiler Settings

A project file is used to identify the external's source code, and specify various compile and link options. Here are a few of the many options specified in the project's Settings dialog:

1) Choose the project type "Code Resource" in the "Target/ 68K Target" pane
2) Set the Creator to CDP3
3) Set the file Type to OEmu for menus or OEtl for tools
4) Set the ResType to XCDD
5) Set the ResID to 100
6) Turn on the "Extended Resource" checkbox to enable multi-segment code resource feature
7) Choose "Standard" for the Header Type popup menu.

There are too many C++ options to enumerate, but all sample projects, libraries and the project templates are set correctly. We recommend you start any external from a sample or a template. If you decide not to, or you accidentally change the settings in you project, compare your projects settings with those of the supplied projects.

Add the file "`ToolStub.rsrc`" or "`MenuStub.rsrc`" to your project. It contains a small code resource of type 'XCOD' and id 50.

Managing A4

As externals are code resources, they introduce some new problems into development that would not exist were they full-fledged applications. The first of these is that code resources use the A4 register for global data. In CodeWarrior this behavior is optional, so to use it requires a small amount of coding. In case you think you might not need global data, you should know that global data includes any global or static variables in your code as well as information the compiler generates to manage multiple segments in a code resource. Since all non-trivial externals require multiple segments you must write code to manage A4. Fortunately very little code is required to do this and if forget to do it your external will be quick to remind you by crashing when you try to use it.

There are two places in your external where you will need to insert code. The first place is in you main function.

```
extern “C” long main (/*parameters as appropriate*/ CallBackPtr
cbp)
{
      RememberA0();
      SetUpA4();
```

```
        MCXMemorizeA4(cbp);

        /* Do your thing here */

        RestoreA4();
}
```

The first two lines save the original A4 register and set it to point to your global variables; the third saves the A4 register so VectorWorks can restore it later. The last restores the old value of A4. RestoreA4 must be called before you exit, so don't return from the middle of the function without calling RestoreA4 first. There is an exception to this rule. When a menu chunk receives either the kMenuCheckHilite or kMenuNotify actions, its segments are not loaded, so do not call RestoreA4. For more information on how all this works, see the CodeWarrior manual.

Note: for this code to compile you will need to #include SetUpA4.h. Since SetUpA4.h uses inline assembly be sure that ANSI conformance is off in your C++ options.

The second place you will need to add code is at the start of any function, other than main, that is called by VectorWorks (for example: all `ForEachObjectProcs`, and `InteractiveDrawProcs`).

```
extern "C" void MyRoutine(/*whatever*/, CallBackPtr cbp)
{
        A4Recaller a4r(cbp);
        /* Any of your own variables */

        /* Do your thing here */
}
```

The A4Recaller variable uses a constructor to restore the value of A4 saved by MCXMemorizeA4, so that the function can use your global variables and call functions in other segments.

Note: You may have noticed that both of the above functions are declared:

```
extern "C"
```

This specifies the way parameters are passed to the function. Since VectorWorks always uses C calling conventions when calling functions in an external you need to declare all such functions (your main function and any callback function) as extern "C". Failing to do so will result in garbage for parameters.

Using Libraries in an External

Using A4 causes another problem. This one is equally as nasty in the errors it causes, but the work necessary to eliminate it need only be performed once. Libraries that ship with CodeWarrior (SANE, ANSI...) were built to be used with applications. This means that they use A5 for global data and inter-segment calls. If you try to use the libraries as shipped your external will crash in strange and unreliable ways. One step we've taken to reduce the risk of this is to adopt a naming convention for libraries compiled for use with externals (we suggest you use it too). Any library compiled with A4 relative offsets is named <Old Name>-A4. If it is a version particular to C++ then '++' is appended to the library name (after the -A4). All supplied projects follow this convention so following it will make compiling projects supplied by Diehl Graphsoft easier. (Note: if you try to compile any of the supplied projects without first creating the new libraries, CodeWarrior will complain that the libraries cannot be found.)

To make a library compatible with externals follow these steps:

- From the Finder, find and create a duplicate of the library. Rename the copy to <Old Name>-A4.
- Double-click on the newly created library. This will automatically launch and open the library.

- Choose “Set Project Type…” from the Project menu.
- Choose the “Code Resource” radio button. This indicates that the project should be recompiled using A4 relative offsets.
- In the “Type” field, type in four question-mark characters (“????”, though any four characters are fine). Type “20” into the ID field. Hit the OK button to exit the dialog.
- Hit the “Continue” button on the alert dialog warning you that all files in the project will need to be recompiled.
- Choose “Make…” from the Source menu. After all the files have been compiled, save the project. This is now a legal file for use in VectorWorks externals.

Remember, appending -A4 to a library name is just a convention and does not guarantee that the library has been compiled correctly. If you are in doubt about a given library you can always open the library and check that the project type has been set to Code Resource.

The SDK includes two AppleScripts to make this process as painless and as error free as possible. The first of these, when executed, will duplicate all the libraries that need to be changed, rename the duplicate to follow the convention, and recompile the copy with the correct settings. This script requires the scriptable finder or MacOS 7.5 or later. If you do not have either of these the script is still useful for it will prompt you to carry out any actions it is unable to perform. Even if you don’t have AppleScript, the script is important because the log file it uses contains the names of all the libraries that need to be updated and what their settings should be.

Note: This script relies on the directory structure of your CodeWarrior development tree being the one the CodeWarrior installer setup. If you have made changes to the folder organization the script may not work. If it fails then you must either reconstruct the original directory structure (by hand or by reinstalling) or you can update all the libraries yourself using the script as a guide.

CallBackPtrs

Another problem with externals is one of location. How do VectorWorks and the external find each other? VectorWorks knows where the external is because all externals are accessed through XCOD 50 of their files. VectorWorks, however, can be anywhere in memory. The mechanism VectorWorks uses to solve this problem is the CallbackPtr (because a CallbackPtr allows an external to “call back” to VectorWorks). A CallbackPtr is a reference to a function internal to VectorWorks that services external requests (A CallbackPtr is actually a structure containing several fields, but its primary service is as a link to VectorWorks). When an external calls a VectorWorks function, that function does not actually service the request. Instead, that function packages the request in a special form and forwards it to VectorWorks using the CallbackPtr you pass it. That is why all VectorWorks functions take a CallbackPtr as the first parameter.

Note: VectorWorks provides, as a parameter, a CallbackPtr every time it calls an external. The CallbackPtr is only guaranteed to be correct until your function returns. Therefore, do not bother to save the CallbackPtr between calls to your external because it may not be valid.

Global Data

Using SetUpA4.h allows for global data, but what global data is for a code resource is not the same as for an application. In an external, global data does not persist between calls to the external. This is the same behavior that applications demonstrate (global data does not persist between executions of the application). For an application this behavior seems correct for applications have clearly defined phases for setup and setdown.

The calls to externals however occur in groups and these calls are logically linked. The first call is the setup phase, the last call is the setdown phase, and every call in-between is a request for action from a

“running” external. This model however is not necessary for the notion of a code resource. It is a model created by the VectorWorks open architecture built on top of the simple code resource mechanism.

To make our model work the external is called repeatedly, each time starting up as if it were the first time. However, since VectorWorks persists for the entire logical lifetime of the external, it can remember state information for the external. The external can use that information to restore itself to a previous state providing it saved the information in the first place.

VectorWorks provides persistent data, to store an external’s state, by two means. The first is that any memory dynamically allocated from an external is allocated from VectorWorks’ heap. That memory will remain allocated until the external frees it or VectorWorks quits. (This means that an external can cause a memory leak that affects the whole application. So take care with your memory management.) The second is the four bytes of storage that VectorWorks reserves for an external (the userData parameter to the main function). This space can be used to store a handle or pointer to memory dynamically allocated by the external.

There is another caveat about global data. Global and static variable are not automatically initialized even if they have a constructor. This is because a code resource is considered to be another function. This means that, unlike an application, there is no startup phase for an external. There is no time that the compiler sets aside to initialize global data. The solution is to avoid using global data, and especially data of non-primitive types. If you must use a custom type for global data (which is not unlikely), then make the global data a pointer or handle, and allocate the object during each call of your external that needs it. Constructors are always called when an object is allocated with new.

Debugging

To diagnose and fix problems with your external, it is possible to debug it at the source level using CodeWarrior. You will be able to set breakpoints, step through your external's source code line by line, break at watchpoints and many other debugging features.

In order to enable the debugging of external code resources, you must disable MetroNub's CloseResFile patch as described in the Metrowerks CodeWarrior release notes. Make the following minor modification to the **MetroNub** file in your "System Folder:Extensions" folder:

- Open the "MetroNub" file using ResEdit
- Edit the "CFRE" 128 resource
- Set its value to 0
- Save the "MetroNub" file and quit ResEdit
- Reboot the machine.

Alternately, the SDK includes the DebugWindow utility (developed by Keith Ledbetter) to display printf style messages from an external.

Index