Chapter5
Writing a plug-in

This chapter describes the steps involved in writing, compiling, and executing a plug-in module. Plug-in modules are written in C++ and can be compiled using the MPW, MetroWerks, or Symantec C++ compilers. No special familiarity with C++ (as opposed to C) language features is required.

5.1 Creating a new plug-in

This section walks you through the entire process of creating a new plug-in. There are five steps in the process:

- Define your plug-in
- Choose a sample file as a starting point
- Write the plug-in
- Build the plug-in
- Test and debug

5.1.1 Defining your plug-in

The first step in writing a plug-in is to define what you want the plug-in to do. This is the first step in any programming project; however, when writing a plug-in module, there are some specific questions which you should ask yourself:

- How does your plug-in interact with the Arrange callback API? Does it create notes? Edit fields? Intercept user actions such as clicking or typing? Which callback functions will it need to use to accomplish its tasks?
- What changes does the plug-in make to the Arrange user interface? Does it add new menu items? Define new formats for the Import and Export dialogs? Change the meaning of mouse clicks when a particular modifier key is pressed?
- How does the user control the plug-in? Does it need a Preferences dialog? Should it be selectable as a Format or Action in the Define Field dialog?
- Does your plug-in make direct calls to the Macintosh operating system or toolbox? Does it display dialogs of its own? Can these be implemented using

the callbacks in the DialogCalls section, or do you need to write your own dialog code?

5.1.2 Choosing a sample file

The Arrange Plug-in Developer's Kit includes a number of sample plug-ins, with source code, that you can use as a starting point for writing your own plug-ins. Of course you can always decide to start from scratch, but it is much easier to begin with a working plug-in that resembles your own, at least in the ways in which it interacts with Arrange. The following sample plug-ins are useful as starting points:

- "Auto-completion", a plugin which ships with Arrange 2.0, shows how to define an "action" selectable from the Define Field dialog, how to monitor editing operations in a field, and how to take an action when the user clicks out of a field.
- "Select Subnotes", another shipping plugin, shows how to add new commands to the menu bar and how to work with note selections.
- "Meeting Maker" shows how to define custom import and export formats.
- "Generic Plugin" is a minimal plug-in shell, with ready-made slots into which your code can be inserted.

Additional sample plugins will become available from Common Knowledge in a future version of the Developer's Kit.

5.1.3 Writing a plug-in

Once you have selected a sample file to use as a starting point, you write a plug-in much as you would write any other piece of code on the Macintosh. Be careful to respect the guidelines in section 4.2.3, "Toolbox calls". Also note that plug-ins compiled using the MPW C++ compiler cannot use global variables, since plug-ins don't have an "A5 world".

To begin with, delete or comment out any sample code which doesn't apply to your task. You may wish to include pieces of code from other sample plug-ins. For example, if you are writing a plug-in which adds menu commands and defines a custom import format, you might want to start with the "Meeting Maker" sample and bring in code from "Select Subnotes" for adding a menu command. All of the sample plug-ins are based on the "Generic Plugin" sample, making it easy to mix and match pieces of the different samples.

Every plug-in has a "ModuleID" which must be unique. Plug-ins may also add new commands to the menu bar or define new import or export formats; all of these things need their own unique IDs, as described elsewhere in this document. You must remember to change the IDs used in the sample files to your own IDs. Blocks of unique IDs can be obtained from Common Knowledge; for internal use only, you can use IDs in the range 0x70000000 to 0x7FFFFFFF. This ID range will never be used by Common Knowledge.

You should remember to look at the sample plugin's .r file to edit any resources which need changing, such as the DITL for the plug-in's about box.

5.1.4 Building a plug-in

This section describes how to compile and link a plug-in. Plug-ins can be built using the MPW, MetroWerks, or Symantec C++ compilers. A description of the build process using MetroWerks or Symantec will be added shortly; for now, we only describe the MPW build process.

5.1.4.1 Building under MPW C++

To build your plug-in using MPW, set the current directory to the directory containing your source code:

```
directory "MyHardDisk:MyPluginSource:"
```

Next, compile each of your source (.cp) files. A typical plug-in will only have a single source file.

```
CPlus -sym on -mbg on -i MyHardDisk:PluginInterfaces: _
      myPlugin.cp -o myPlugin.cp.o -mf -b2
```

Replace "`MyHardDisk:PluginInterfaces`" by the folder in which you have placed the interface (.h) files supplied with the Arrange Plug-in Developer's kit.

Next, compile your plug-in's resource:

```
Rez -d REZ "{RINCLUDES}"types.r "{RINCLUDES}"systypes.r _
    -include MyHardDisk:PluginInterfaces: _
    myPlugin.r -o MyPlugin
```

Finally, run the linker. (WARNING: you must tell the linker to generate an 'MDcd' resource with the same ID as the 'MDdf' resource in your .r file. In the "Generic Plugin" sample, this is -32768 as shown below. Other sample plugins use other IDs. If you generate an 'MDcd' resource with the incorrect ID, Arrange will not be able to find your plug-in's code, and you will get a debugging message when Arrange starts up.)

Depending on what parts of the Macintosh Toolbox your plug-in uses, you may have to link with

additional MPW libraries such as {Libraries}Interface.o or {CLibraries}StdCLib.o.

```
Link myPlugin.cp.o _

     MyHardDisk:PluginLibraries:PluginLibrary.lib _

     -m OurModuleRoot -c 'TSLA' -t 'Aplg' -sym on _

     -rn -rt MDcd=-32768 -ra =resLocked,resPreload _

     -sn STDCLIB=Main _

     -o MyPlugin
```

You are now ready to test your plug-in, as described in the next section.

5.1.5 Testing and debugging

To test your plug-in, drag it into the Arrange "Plug-ins" folder and reboot Arrange. If you were already running Arrange with a previous build of your plug-in, you will have to quit Arrange before overwriting the old build.

Now try out your plug-in. Select every menu item that it creates or overrides. Check that it adds an item to the About Plug-ins menu and that this works properly. Verify that its menu items are disabled when they don't apply (you should do this in your AboutToMenu hook). Also verify that your plug-in doesn't interfere with the normal operation of Arrange. For example, if you are defining a new import format, verify that existing import formats still work properly.

Information about debugging plug-ins and dealing with crashes is given in the next chapter.

5.2 Naming and registering your plug-in with Common Knowledge

When you have finished your plug-in, contact Developer Services at Common Knowledge to register your plug-in.