The Macintosh Developers Guide to the Novell SDK

Richard Kendall Wolf <richwolf@uic.edu>
Paul Neumann <pauln@uic.edu>

You don't like it, we don't like it—none of us likes it really—but, painful as it is, we have to admit the truth. And the truth is that the dominant network server platform found nearly everywhere today isn't AppleShare running on a Macintosh. In fact, if you work in a mixed environment of PCs and Macs, then the chances are very good that you use the services provided by Microsoft's Windows NT Server or Novell's NetWare to share and access local network resources. That's just the way life is.

As a Macintosh developer, Windows NT doesn't present you with any particular challenge since a server running NT behaves just like a server running AppleShare—at least from the point of view of your code. The flip side for you is that, in an NT world, you don't share a common SDK with your PC colleagues. And that makes it difficult for you to create client/server solutions that allow Macs to become full partners with PCs in your enterprise.

Novell's NetWare, however, offers you an opportunity to code on an equal footing with your PC colleagues. While a NetWare server can behave just like an AppleShare server, Novell has provided its Macintosh clients with the ability to become full peers in a NetWare environment. The trade-off this time is that to you must learn to code with the Novell SDK.

Therefore, what we hope to do in this paper is present you with the basics of the Novell SDK as seen from a Macintosh developer's point of view. Of course it will not be possible for us to present the entire SDK, or even to present a small part of it in any detail (after all, that would the subject for an entire book). But we do hope to give you a feel for how you would go about working with your PC colleagues to create cross-platform client/server applications using the SDK. Along the way we hope to persuade you that there are compelling reasons for becoming more acquainted with the Novell SDK. In addition, we'll show you an example of code using the Novell SDK and that we feel you will find both instructive and interesting. The code example is a utility that looks for and runs a login script maintained by a NetWare administrator. Our client will mimic what Novell's PC clients do, but the difference for our client is that ours can run AppleScript.

Good Reasons to Skip Our Paper

We know—you probably don't care much about this stuff. In fact, we'll bet you'd rather be reading about C++ pitfalls or Java development. And we agree—we weren't sure if writing a paper about the NetWare SDK would really be worthwhile. But after some thought, we figured that many of you probably live in environments rather like our own—environments where PCs are the predominant platform and PC networking solutions are ubiquitous. In environments like these, you have to create code that helps make Macs full partners with PCs. Luckily, if you live in a Novell environment, you have the ability to do just that.

However, you may have some of the same legitimate concerns that we did when we began our paper.

Isn't Novell Slowly Dying Away?

You may have heard from your PC colleagues that Novell is moving slowly on its way to oblivion. But the surprising fact is that Novell remains the fifth largest software manufacturer in the world with over 79 million customers.[1] In addition, Novell's share of the server operating system market is still the highest at 41%.[2]

Are Macintosh Users Really NetWare Users?

If you come from a predominantly Mac environment, you may feel that every important server for your users is an AppleShare server—and everything you need to know about AppleShare you learned in *Inside Macintosh: Files* and *Inside Macintosh: Networking*.

But again, the surprising fact is that many Macintosh users really do care about the services offered by NetWare. We asked AG Group's networking troubleshooting list[3] if any of its members were currently living in a NetWare world and whether they'd like to get more developer support. The following are typical of the many responses that we received.

> **Yes, we do live in a NetWare environment—we have over 130 NetWare servers actually. I'd** love **to have some more tools!**
>
> In my role as the main residential networking support person, I have been often frustrated by the lack of good end-user tools for the Mac.

1 http://www.novell.com/corp/.

2 *International Data Corporation (http://www.idc.com) estimates that as of September 30, 1997 there were some 9.4 million server operating systems in use worldwide. Based on data published by IDC in January, 1998, Novell holds 41% of the market.*

3 *mailto: net-troubleshooting@lists.aggroup.com. This list specializes in Macintosh networking issues.*

And:

> I'm the network administrator for Apple's ad agency, based in Los Angeles. Across all our North American offices, we have an installed base of 1000+ Macs (and more in our European offices)—and all use NetWare fileservers.

While our survey is by no means scientific, we hope it changes how you feel about developing software for Novell's Macintosh client.

The News from WWDC

You may have heard that Novell no longer supports its Macintosh client software and SDK. While that's not completely true—Novell does support its Mac client—it is true that Novell hasn't been doing a very good job lately. But the good news from this year's WWDC is that Novell has made an agreement with a third party, Prosoft, Inc., to take over support for its Macintosh client and SDK—and Prosoft seems bent on providing the best possible environment for Macintosh users and developers. You can find out more about Prosoft by checking out the references at the end of our paper.

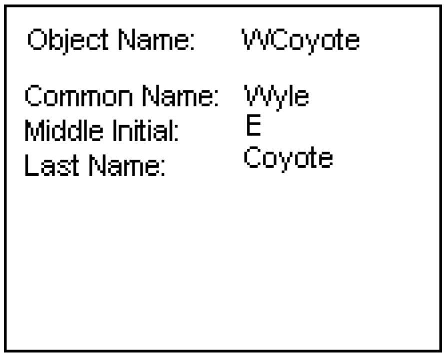
A Little Background Information

Simply stated, NetWare is Novell's network operating system capable of supporting an extremely wide variety of host operating environments including DOS, Windows 3.1, Windows 95, Windows NT, OS/2, UNIX, and the Mac OS. NetWare provides network file and print services, just as you'd expect from a network operating system. In addition, it also provides a speedy web server, routing capabilities, and a full compliment of administrative tools. But most importantly, it contains a very powerful mechanism for creating a directory of network services. In fact, directory services management is at the heart of nearly all NetWare environments. So in order to understand how NetWare environments work, we have to spend some time getting to know how NetWare Directory Services work.

NetWare 4.x and the NDS Tree

Novell introduced its Directory Services architecture, or NDS, when it released NetWare version 4.0. NDS manages a hierarchical database of objects that exist within a NetWare 4.x environment. NDS objects include things like servers, users, printers, print queues, volumes—nearly anything that can exist within a Novell network can be represented by an NDS object. One of the key features of NDS is that once you, as a network user, are granted access to an NDS-managed network, you may access whatever objects for which the proper permissions have been granted you no matter where those objects exist in the network.

Since NDS is a database, each object it contains has certain properties and each of those properties is associated with a set of values. For example, let's say there is a network user whose name is Wyle E. Coyote. As a Novell administrator, you could set up a user object for Wyle and call it WCoyote. You would add this user object into the NDS database along with its properties and associated values. An example of a property is WCoyote's first name and, obviously, its associated value is Wyle. In addition, WCoyote's middle initial property would be set to E and his last name property would be set to Coyote. NDS stores additional property / value associations in Wyle's user object, such as any public keys he needs for authentication, telephone numbers where he can be reached, login timestamps, as well as lots of other information—and user objects are only one type of object maintained by NDS—different kinds of objects, such as print queues and network volumes, contain different properties and associated values.



Object Name:	WCoyote
Common Name:	Wyle
Middle Initial:	E
Last Name:	Coyote

Figure 1: *Sample NDS objects containing various properties and values*

In a large organization, there may be tens of thousands of users—not to mention hundreds of servers. Managing such a large number of network objects can be a difficult task especially if those objects aren't organized somehow. Therefore NDS organizes objects hierarchically by grouping them into trees, organizations, and organizational units.

NDS trees generally contain the whole collection of objects for any particular environment. Typically, even very large institutions will choose to implement only one tree for the entire organization. In any NDS tree, you will find a series of organizational containers. For example, an institution might have both a marketing group and an engineering group. A logical way to split up the institution's tree would be to put user objects associated with the marketing group into a "Marketing" container and the engineers' user objects in an "Engineering" container.

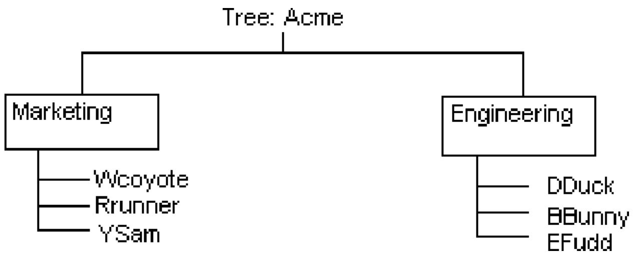


Figure 2: *A simple NDS tree*

Think of organizations and organizational units as types of network folders. Inside folders (organizations, organizational units) you keep files or subfolders (objects). In any NDS tree, however, only top level folders can be organizations. Subfolders located inside organizations must be organizational units. Organizations hold organizational units, which may, in turn, hold other organizational units.

Naming in NDS

Novell has worked out a naming scheme for referring to objects within an NDS tree. Continuing with our example above, the fully qualified name for Wyle E. Coyote's user object would be CN=WCoyote.O=Marketing. The object name is preceded with a CN=, organization names are preceded with O=, and organizational unit names are preceded with an OU=. You separate parts of any name with periods. Using another example, my fully qualified name at UIC is CN=PaulN.OU=Comp.O=UIC. Names are read left-to-right, leaf-to-root. For example, my user object, PaulN, is contained within the Comp organizational unit which, in turn, is contained within the UIC organization. The NDS tree root is not named.

Now specifying fully qualified object names requires a lot of typing! Fortunately, Novell has provided a shorthand way of specifying NDS objects. Just leave off the CN=, OU=, and O= parts of any fully qualified name—NetWare will work out what the other parts of the name must be. So, using this shorthand, my user object can be specified PaulN.Comp.UIC. The first part of this shorthand name is referred to as its common name (c.f., CN= in the fully qualified example) and the last part is referred to as its context.

Still with us? Okay, we've gotten through all the things you need to know about NetWare to get through the rest of what we have to say. Obviously, there is a lot more to know about NetWare than what we've said here, so we've given you some references to check out at the end of our paper.

The Novell SDK

Installing and using the Novell SDK is actually pretty painless. But before we can begin getting our hands dirty with it, we'll have to consider some preliminary issues. The first of these issues will be setting up our development Macintosh so that it can access our local NetWare environment. And in order to do that, we'll have to get and install the NW Client for the Mac OS.

About NW Client for the Mac OS

Getting the NW Client for the Mac OS is easy—Novell distributes its client software freely over the web.[4] The current version of the NW Client for the Mac OS is version 5.11. Since we're running Mac OS 8, we also have to remember to download the client patches Novell provides for Mac OS 8 and install them as well. If you're interested in installation specifics, we refer you to Novell's web site.

If you live in Novell world like we do, then you'll notice that the NW Client for the Mac OS offers you a lot of nice features. One of the nicest is a true NCP (Novell Core Protocol) requestor, just like the one Novell's PC clients have. In addition, the NW Client for the Mac OS can make NCP requests both through AppleTalk and IPX/SPX. We'll concentrate on using the NCP

4 http://www.support.novell.com/products/nwcmc511/.

requestor directly through IPX / SPX because of the wider participation we can have using it in our local NetWare environment.

Installing the SDK

Installing the Novell SDK is simple, just copy the “NW Client for Macintosh SDK” folder onto your Mac’s hard drive from the Novell SDK CD.[5] You can place the SDK folder wherever you like—if you do a lot of NetWare coding, you may want to place it so that your development environment can access it simply.

Inside the “NW Client for Macintosh SDK” folder, you’ll find header files, libraries for different compilers (including Metrowerks’ CodeWarrior—the IDE we’re using), and example code.

SDK 7

Like Apple, Novell releases SDK updates from time-to-time. And with each new SDK release Novell increments the SDK’s release number. As of this writing, the current SDK release is SDK 15. To work with the Macintosh, you will need to obtain SDK release 7 or any subsequent release. Since the Macintosh SDK hasn’t changed since release 7, you can get whatever release you like, but we recommend obtaining release 7 specifically if you can.[6] The reason why we’d like you to get your hands on release 7 is because release 7 contains the most (and best) Macintosh documentation of any SDK release.

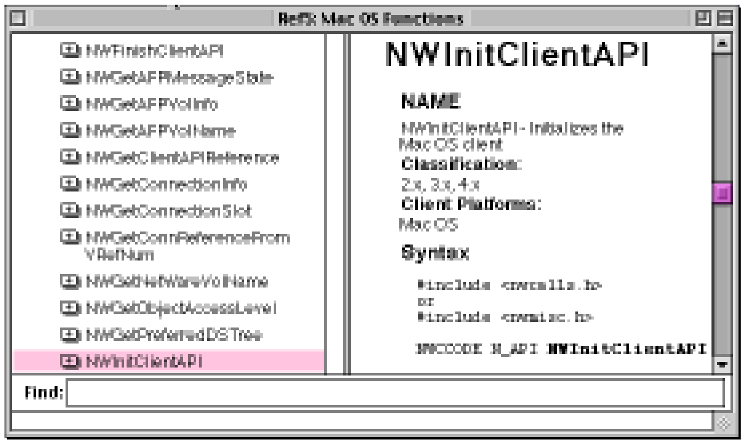
And, just so you know, Novell specifically assigns its Mac OS SDK a version number unrelated to the SDK release number—the version released with SDK 7 (and subsequent SDK releases) is version 1.1.

A Really Lousy Application

We are cramming a lot of information into this paper, but you will surely need to look something up that we won’t get a chance to talk about here. To do that, you will have to get familiar with the worst application we’ve ever used—DynaText. DynaText is Novell’s answer to DocViewer—that is, an application you can use to look up anything you need to know about the Novell SDK. If it is any consolation, DynaText is equally lousy on both PCs and Macs.

However, if recent versions of DynaText are any indication, things may be improving dramatically in the future. Recent versions of DynaText can access much more than the bare facts about the Novell API including things like tutorials on how to use the Novell SDK and specific technical information regarding which API calls and libraries to use when you have a specific purpose in mind. You can even use it to see where Novell is headed in the future by checking out its “futures” section.

You’ll find DynaText on the SDK CD.



NWInitClientAPI

NAME

NWInitClientAPI - Initializes the Mac OS client

Classification:

2.x, 3.x, 4.x

Client Platforms:

Mac OS

Syntax

```
#include <nwcalls.h>
or
#include <nwmisc.h>

NWCCODE N_API NWInitClientAPI
```

Figure 3: *DynaText*

5 If you don’t have the CD, you can get the SDK free from Novell. Just register yourself at http://developer.novell.com/cgi-bin/nvolve/register? (you need to include the question mark).

6 We complained a lot and Novell offered to send us a free CD. We’re not suggesting that our strategy will work for you—we’re just saying what worked for us and from that we’ll let you draw your own conclusions.

How Novell's SDK Works with Your Application

When you link Novell libraries into your application, your application may contain the actual routines that perform the services you require or stubs that call code loaded into the system by various Novell extensions during bootup. Novell seems to want to strike a balance between the speedy execution of your code and your code's size. The nice thing for you, as a developer, is that your don't need to know which routines are stubs and which aren't.

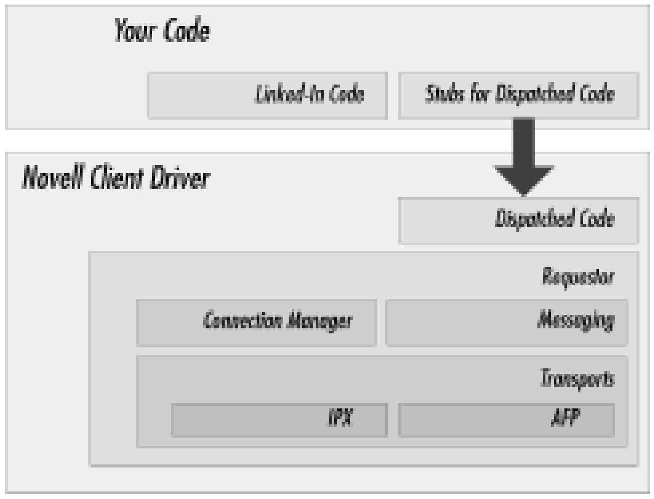


Figure 4: *How your code works with Novell's libraries*

Basic Novell Types

We'll have to use some basic Novell types in our code. It's important that you use these types rather than the simple types you know because your code may not work if you decide to port it to another platform. For your reference, here are brief descriptions of the major types you'll encounter in typical code.

Type Name	What It Is	
nint8	8-bit, signed integer	
nuint8	8-bit, unsigned integer	
nint16	16-bit, signed integer	
nuint16	16-bit, unsigned integer	
nint32	32-bit, signed integer	
nuint32	32-bit, unsigned integer	
nint	singed integer	
nuint	unsigned integer	
nreal32	32-bit real	
nreal64	64-bit real	
nreal80	80-bit real	
nreal	80-bit real	
nfixed	32-bit fixed-point number	
nbool8	8-bit boolean	
nbool16	16-bit boolean	
nbool32	32-bit boolean	
nbool	boolean	
nflags8	8-bit NetWare flags	Same as nuint8
nflags16	16-bit NetWare flags	Same as nuint16
nflags32	32-bit NetWare flags	Same as nuint32
nstr8	8-bit character	
nstr16	16-bit character	
NWRCODE	Return code	Same as nint32
nptr	Pointer	
npproc	Pointer to a function	
pnptr	Generic pointer to a pointer	
pnpproc	Generic pointer to a pointer to a function	
p + type	Pointer to type	
p + p + type	Pointer to a pointer to type	

For pointers, p is generally prepended to a type name to derive a pointer to that type. For example, pnstr8 is a pointer to an 8-bit character. We'll encounter other types in our code, but we'll describe them in detail as they arise.

Reading and Writing from Stream Objects

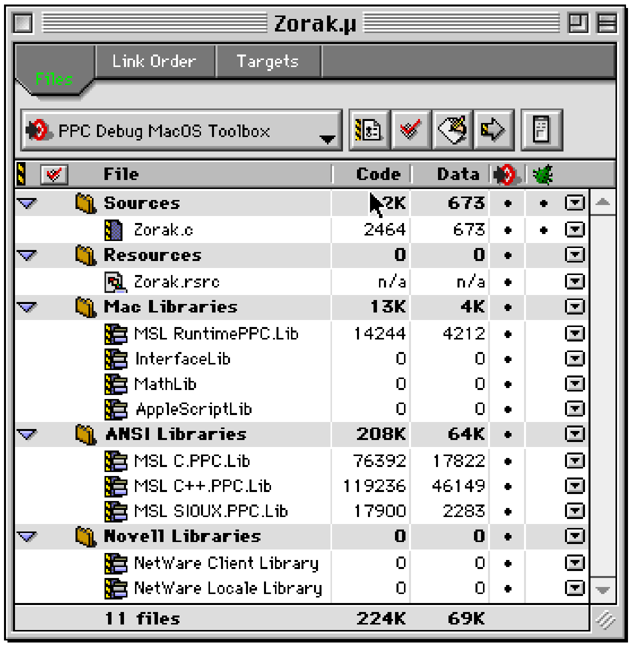
Recall that we discussed NDS objects earlier. We said that NDS objects stored hierarchical data composed of properties and values. One property we'll look at later is a user object's login script (the script that gets run whenever a client logs into an NDS tree). The login script's value is stored as a stream of bytes. Reading those bytes is simple in every environment—except in the UNIX and Macintosh environments. We will see later that we need to make a special set of API calls to read and write stream data associated with NDS objects. We're mentioning this now because this is one place where Novell's API for the Macintosh is really different than the APIs used by other platforms. SDK 7 describes this, and other similar differences, in detail. In general, the Macintosh API set differs very little from those used by other platforms—but there are a handful of significant differences of which you should be aware.

Okay, we're done describing the SDK. It's time for code.

Code Example:

Our code example is going to be an application that queries our local NDS tree and obtains and executes any login scripts it finds associated with our login object. This mimics what Novell's PC clients do automatically—with a difference—the login scripts we access can be coded using AppleScript (but you could also use any other scripting environment, if you prefer).

The first thing we'll do for our code is setup a basic Mac OS Metrowerks project. In addition to the usual Mac OS libraries, we're going to include appropriate Novell libraries. Since we're setting up a PowerPC project, we'll include Novell's PPC libraries in our project file (if we were creating a 68K application or system extension, we could make use of the A5 and A4 libraries Novell also provides).



File	Code	Data		
Sources	**2K**	**673**	●	●
Zorak.c	2464	673	●	●
Resources	**0**	**0**	●	
Zorak.rsrc	n/a	n/a	●	
Mac Libraries	**13K**	**4K**	●	
MSL RuntimePPC.Lib	14244	4212	●	
InterfaceLib	0	0	●	
MathLib	0	0	●	
AppleScriptLib	0	0	●	
ANSI Libraries	**208K**	**64K**	●	
MSL C.PPC.Lib	76392	17822	●	
MSL C++.PPC.Lib	119236	46149	●	
MSL SIOUX.PPC.Lib	17900	2283	●	
Novell Libraries	**0**	**0**	●	
NetWare Client Library	0	0	●	
NetWare Locale Library	0	0	●	
11 files	**224K**	**69K**		

Figure 5: *Basic Metrowerks project with Novell libraries*

Now that we have a project file set up we can turn to writing code.

Novell likes to hype the idea that its development environment is cross platform. Therefore, as developers, we share the very same set of header files with our PC colleagues. And since PCs and Macs are different kinds of platforms, Novell has to provide us some way of specifying particulars—such as how big doubles and ints are. The way they do that is simple—they get us to specify the kind of platform that we are developing for before we include any of their header files in our code. In fact, no matter which platform you develop for, you must tell the Novell header files what it is before you include any of them in your code. So let's go ahead and do that. N_PLAT_MAC is Novell lingo for "We're a Mac."

```
/* Tell Novell what kind of platform we are */
#define N_PLAT_MAC
```

The following are the basic Novell header files to include in your code if you want to get access to the entire Novell API set.

```
/* The basic set of Novell header files */
#include "nwcalls.h"
#include "nwnet.h"
#include "nwlocale.h"
```

Next comes a global used by the Mac OS's scripting components architecture. Since we want to introduce you to the Novell API, we're going to skip a big, long discussion of scripting components. If you want to find out how our code manages to compile and execute a series of scripts, we'll point you in the direction of *Inside Macintosh: Interapplication Communication*. The code we, um "borrowed," is fully described by Apple in Chapter 10. If you're worried about reading hundreds of pages of *Inside Macintosh*, don't sweat it—you can start reading straight from Chapter 10 and by the time you finish the first ten pages, you'll completely understand how "our" code works.

```
ComponentInstance  gScriptingComponent;
```

We'll declare one last global variable that will hold the name of the user object whose scripts we're interested in obtaining.

```
charuserObjectName[ 256 ];
```

Our main routine is simple—initialize the Toolbox and run scripts.

```
void main(void)
{
  NWCCODE error = noErr;

  Initialize();

  /* Get and execute all our scripts */
  error = DoLoginScripts();
  if (error == noErr)
    return;

  return;
}
```

Okay, now we're ready to tackle the good stuff. DoLoginScripts will get and execute login scripts for our user object. The reason the name is plural is because our login object can have multiple scripts associated with it.

```
NWDSCCODE DoLoginScripts(void)
{
  NWDSCCODE      error;
  NWDSContextHandle    context;
  nuint32      flags;
  nstr8        contextName[MAX_DN_CHARS+1];
```

The following is one area where the Macintosh API differs from all others. In order to get things going we have to make this call. But be careful!—a lot of example code from Novell assumes DOS or Windows clients and it initializes itself using NWCallsInit which does not work on the Mac! Basically, though, it has the same purpose in mind—initialize the API for use by our code.

```
  error = NWInitClientAPI();
  if (error) {
    DoError("Call to NWCallsInit failed",
      error);
    return(error);
  }
```

Before we can begin getting and setting NDS information, such as getting the login scripts associated with a particular user object, we have to create a context handle to use when calling NDS routines. The context handle returned by NWDSCreateContextHandle points to a structure whose contents contain our program's current working context (remember contexts from our discussion earlier?). However, the context structure the handle points to contains lots more information than the context name alone. In fact, we think that Novell probably could have come up with a better name for what it calls a "context handle." At any rate, you mustn't confuse the handle returned by NWDSCreateContext with the handle returned by Mac OS routines such as NewHandle—your compiler will probably complain if you try—but hey, if you're desperate, you might be tempted to try some coercion. We know how you feel—but don't try it here.

```
  error =
    NWDSCreateContextHandle(&context);
  if (error) {
    DoError("NWDSCreateContext failed",
      error);
    return(error);
  }
```

Once we have a context handle, we'd like to take a look at some of the information to which it points and NWDSGetContext is the routine for that. Getting the information you want is simple—just pass NWDSGetContext a constant whose value is the particular information you are interested in. Right now, what we'd like to know is how we're currently referring to objects in our local NDS tree. We might be referring to them by their types, for example, or by using fully qualified names. It turns out that we are going to have to ask to have NDS names fully qualified—this will make our lives simpler because then we will be able to refer to objects from the root of our local tree. So, we're going to get the current set of flags associated with our current context, modify a couple of them, and reset our context accordingly. By the way, notice that from now on if we encounter some kind of problem with our calls to Novell Directory Services, we are going to free our context handle before we exit this routine.

```
  error = NWDSGetContext(context,
    DCK_FLAGS, &flags);
  if (error < 0) {
    DoError("NWDSGetContext failed",
      error);
    goto terminate;
  }
```

Okay, here is the place where we change those context flags. Setting typeless names means we don't have to prefix everything with those "OU=" and "CN=" tags. Canonical naming means that we always have to give objects their full names—"rich.mac.uic"—from the root of our local NDS tree. The nice thing is that we will get full names back whenever we ask for them. Later we'll see that this makes our code a lot simpler.

```
  flags |= DCV_TYPELESS_NAMES;
  flags &= ~DCV_CANONICALIZE_NAMES;
```

Guess what we're going to do now? Yup, we're going to reset our context information using NWDSSetContext passing our new value for the context flag information.

```
  error = NWDSSetContext( context,
    DCK_FLAGS, &flags);
  if (error < 0) {
    DoError("NWDSSetContext failed",
      error);
    goto terminate;
  }
```

Now that we will get all object names the way we want, let's grab our program's current working context in that format. This is why we hate Novell's nomenclature—our login object has a context within the tree and our program has a context handle that points to a specific place in the tree. Right now, the place where our program's context handle points to is the place where our user object lives.

```
error = NWDSGetContext(context,
  DCK_NAME_CONTEXT, contextName);
if (error < 0) {
  DoError("NWDSGetContext failed", error);
  goto terminate;
}
```

Before we start getting actual script data, let's set our program's context handle so that it points to the root. We will have to make our program look in multiple places within NDS to get scripts that belong to us. It will be simpler to specify those places by using the root of our local tree as a reference point

```
error = NWDSSetContext(context,
  DCK_NAME_CONTEXT, DS_ROOT_NAME);
if (error < 0) {
  DoError("NWDSSetContext to root failed",
    error);
  goto terminate;
}
```

Finally, we'll find out just who we are by getting the name of our user object.

```
NWDSWhoAmI(context, userObjectName);
```

PC clients can access no fewer than four different scripts when logging into an NDS tree and they get those scripts in a specific order, so we will have to mimic that order in our program. Now it turns out that the last script a PC client can get is a default script provided by Novell. As NetWare administrators, we cannot modify that script, and since it is not in a format that Macs can readily understand, we'll skip it for simplicity. It's pretty dull anyway. Instead, we'll concentrate on the other three scripts and their execution. What we'll do is obtain each script, in its proper sequence, try to run it, then move onto the next.

The first script our code tries to access is the script associated with our user object's context.

```
error =
  GetObjectScript(context,contextName);
  if (error) {
    DoError("GetObjectScript failed",
      error);
    goto terminate;
  }
```

The second script our code tries to access is the script associated with our user object's profile object (an NDS object can point to another, something like an alias).

```
  error = GetUserProfileScript(context,
    userObjectName);
  if(error) {
    DoError("Attempt to read the objects
      script name failed", error);
    goto terminate;
  }
```

The final script our code tries to execute is the script associated directly with our user object.

```
error =
  GetObjectScript(context, userObjectName);
  if (error) {
    DoError("DisplayLoginAttribute failed",
      error);
    goto terminate;
  }

  terminate:

  NWDSFreeContext(context);

  return(error);
}
```

Let's turn to GetObjectScript which, hopefully, will seem pretty straightforward to you after a short glance. We simply pass it a context handle and the name of an object that has a script associated with it. Since the script property of any object is a stream of bytes to be read and interpreted, we examine its value as if it were a file. Recall earlier that we said that we would need to make special calls to the Novell API in order to do that. However, the calls that we need to make are not that unusual.

```
/* Open stream object associated with a login script */
  error = NWDSOpenStream(dContext,
    objectName, "Login Script",
    (NWDS_FLAGS) 1,
    (NWFILE_HANDLE *)&fHandle);
  if (error) {
    DoError("NWDSOpenStream failed",error);
    return(1);
  }

  /* Read stream data */
  error = NWReadFile(fHandle, 4095,
    (unsigned long *)&readsize,
    (unsigned char *)*scriptData);

  /* Close steam data file */
  NWCloseFile(fHandle);
```

Essentially, we treat NWFILE_HANDLEs as we might FILE pointers in the ANSI C libraries (they don't mix!—we're simply saying that you can think of them similarly).

One unusual thing you may be wondering about is why we're taking special care to get rid of newline characters in GetObjectScript. The reason is that scripting components on the Macintosh will not handle them properly (in AppleScript you will get script compilation errors). So we take the time now to remove them.

The final two routines we'll look at effectively do one thing—retrieve the script associated with our user object's profile object. That statement may sound somewhat odd, but NDS allows us to reference one object in a tree from another (essentially making one object a pointer to another). In GetUserProfileScript, we first determine what profile is associated with our user object—which turns out that this is one of our user object's properties. Once we find our profile object, we'll ask it to show us the script associated with it in DisplayAttributes.

GerUserProfileScript may look complex, but it really isn't if you know how parameter blocks work on the Macintosh. pBuf_T is a pointer to an NDS data buffer. We can stuff a pBuf_T with values to request information about NDS object attributes (attrBuf) and supply an empty pBuf_T to retrive it (outBuf). Once you understand this, the code reveals its purpose clearly. After GerUserProfileScript discovers our user object's profile object (through its profile attribute), DisplayAttributes retrieves the profile's script and passes it to GetObjectScript, which we have already looked at in detail.

More Information, References, Etc.

URLs

General information about Novell and its products can be found at Novell's web site: http://www.novell.com.

Specific information about Novell developer programs and services can be found at Novell DeveloperNet: http://developer.novell.com.

Information about Prosoft and its services can be found at Prosoft's web site: http://www.prosofteng.com.

Books

There are lots of good books available that describe NetWare and NetWare environments, including specifics on how NetWare Directory Services work. One such book is *CNE 4 Short Course* by Dorothy Cady, Drew Heywood, and Debra Niedermiller-Chaffins, New Riders Publishing, 1995.

Unfortunately, there are virtually no good books that describe how to do NetWare programming. The best (and, apparently, only) title is NetWare NLM Programming by Michael Day, Michael Koontz, and Daniel Marshall, Novell Press / SYBEX Inc., 1993. To make matters worse, the last time we checked, this book was out of print. The book has the NLM developer in mind (NLMs are NetWare Loadable Modules—the name Novell gives to server executables), but there is a lot of information in it of general interest to anyone who plans to use the Novell SDK.

To get an explanation on how our code was able to compile and execute plain text as a script on the Macintosh, see *Chapter 10, Scripting Components, Inside Macintosh: Interapplication Communication*, Apple Computer Inc. / Addison-Wesley, 1993.

All Our Code

```
/* Zorak.c */
/* ------- */
/* Run login scripts associated with a Novell user object */

/* Tell Novell what kind of platform we are */
#define N_PLAT_MAC

/* Includes */

#include <stdio.h>
#include <string.h>

/* The basic set of Novell header files */
#include "nwcalls.h"
#include "nwnet.h"
#include "nwlocale.h"

/* Constants */

#define kNil 0L/* Generic nil pointer */
#define kAlertID 128 /* Alert ID to
```

```
    display errors */
#define kTimeOut 10240 /* Timeout value
    to use when running scripts */

/* Globals */

/* Needed for the scripting components architexture */
ComponentInstance  gScriptingComponent;
/* The user object whose login scripts we're interested in */
char      userObjectName[ 256 ];

/* Prototypes */

void Initialize(void);
NWDSCCODE DoLoginScripts(void);
NWDSCCODE GetObjectScript(
  NWDSContextHandle dContext,
  pnstr8 objectName);
NWDSCCODE DisplayAttributes(
  NWDSContextHandle dContext, pBuf_T buf);
NWDSCCODE GetUserProfileScript(
  NWDSContextHandle dContext,
  pnstr8 objectName);
OSAError RunScript(Handle scriptText);
void DoError(char *errorString,
  short errorCode);

void main(void)
{
  NWCCODE error = noErr;

  Initialize();

  /* Get and execute all our scripts */
  error = DoLoginScripts();
  if (error == noErr)
    return;

  return;
}

/* Initialize the Toolbox */
void Initialize(void)
{
  /* Initialize all the needed managers. */
  InitGraf(&qd.thePort);
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs(nil);
  InitCursor();

  return;
}

/* No frills error reporting */
void DoError(char *errorString,
  short errorCode)
{
  Str255 alertString;

  sprintf((char *) alertString,
    "%s:\t%04X.", errorString, errorCode);

  c2pstr((char *) alertString);

  ParamText(alertString, "\p", "\p", "\p");

  Alert(kAlertID, kNil);

  return;
}

/* Initialize Novell APIs and run login scripts associated with our
user object */
NWDSCCODE DoLoginScripts(void)
{
  NWDSCCODE error;
  NWDSContextHandle  context;
  nuint32   flags;
  nstr8     contextName[MAX_DN_CHARS+1];

  /* Initialize the NW API */
  error = NWInitClientAPI();
  if (error) {
    DoError("Call to NWCallsInit failed",
      error);
    return(error);
  }

  /* Create a context handle for use by later calls */
  error =
    NWDSCreateContextHandle(&context);
  if (error) {
    DoError("NWDSCreateContext failed",
      error);
    return(error);
  }

  /* Get the current directory context flags so we can modify
them */
  error = NWDSGetContext(
    context,   /* -- Context Handle */
    DCK_FLAGS, /* -- Key            */
    &flags);   /* -- Context Flags */
  if (error < 0) {
    DoError("NWDSGetContext failed",
      error);
    goto terminate;
  }

/* Turn typeless naming on */
/* Turn canonicalize names off -- this means we will get full
names */
  flags |= DCV_TYPELESS_NAMES;
  flags &= ~DCV_CANONICALIZE_NAMES;

  /* Set the directory context flags so they take effect */
  error = NWDSSetContext(
    context, /* -- Context Handle */
    DCK_FLAGS, /* -- Key            */
    &flags); /* -- Set Flag Value */
  if (error < 0) {
    DoError("NWDSSetContext failed",
      error);
    goto terminate;
  }

  /* Now find out what Context we are logged in under ... */
  error = NWDSGetContext(
    context,   /* -- context Handle  */
    DCK_NAME_CONTEXT,  /* -- key      */
    contextName);  /* -- value returned */

  /* Reset it to be the [ROOT] context ... */
  error = NWDSSetContext(context,
    DCK_NAME_CONTEXT, DS_ROOT_NAME);
  if (error < 0) {
    DoError(
      "NWDSSetContext to root failed",
      error);
    goto terminate;
  }

  /* Find out whose user object we're interested in */
  error = NWDSWhoAmI(context,
    userObjectName);
  if (error < 0) {
    DoError(
      "NWDSSetContext to root failed",
      error);
    goto terminate;
  }

  /* Get the login script associated with our context */
  error = GetObjectScript(context,
    contextName);
  if (error) {
    DoError("GetObjectScript failed",
      error);
    goto terminate;
  }

/* Get the login script associated with our user profile object */
  error = GetUserProfileScript(context,
    userObjectName);
  if(error) {
    DoError("Attempt to read the objects
      script name failed", error);
    goto terminate;
  }

  /* Get the login script associated with our user object */
  error = GetObjectScript(context,
    userObjectName);
  if (error) {
    DoError("DisplayLoginAttribute failed",
      error);
    goto terminate;
  }

  terminate:

  NWDSFreeContext(context);

  return(error);
}

/* Run script associated with NDS object name and context
handle */
NWDSCCODE GetObjectScript(
  NWDSContextHandle dContext,
  pnstr8 objectName)
{
  NWDSCCODE    error;
  NWFILE_HANDLE  fHandle;
  long     readsize;
  Handle   scriptData;
  unsinged char  *temp;
  long     i;
  long     j;

  /* Open stream object associated with a login script */
  error = NWDSOpenStream(dContext,
    objectName, "Login Script",
    (NWDS_FLAGS) 1,
    (NWFILE_HANDLE *)&fHandle);
  if (error) {
    DoError("NWDSOpenStream failed",error);
    return(1);
  }

  /* Reserve room for script data read from the stream */
  scriptData = NewHandleClear(4096);

  HLock(scriptData);

  /* Read stream data */
  error = NWReadFile(fHandle, 4095,
    (unsigned long *)&readsize,
    (unsigned char *) *scriptData);

/*Copy data to temporary storage, but ferret out newlines first*/
  temp = (unsigned char *) malloc (4096);
  for (i = 0, j= 0; i <readsize; i++)
    if ((*scriptData) [i] != '\n') {
      temp[ j ] = (*scriptData)[ i ];
      j++;
    }

  /* Resize stream data handle */
  SetHandleSize(scriptData, j - 1);
  error = MemError();

/* Put data from temporary storage back into stream data
handle */
  for (i = 0; i <j; i++)
    (*scriptData)[ i ] = temp[ i];

  /* Run the script pointed to by the stream data handle */
  RunScript(scriptData);

  HUnlock(scriptData);

  /* Close steam data file */
  NWCloseFile(fHandle);

  return 0;
}

NWDSCCODE GetUserProfileScript(
  NWDSContextHandle dContext,
  pnstr8 objectName)
{
  NWDSCCODE  cCode;
  pBuf_ToutBuf = NULL, attrNames = NULL;
  nint32iterHandle = -1L;
```

/ Allocate a buffer to hold the names of the attributes in which we're interested */*

```
  cCode = NWDSAllocBuf(
    DEFAULT_MESSAGE_LEN, /* Buffer Size
                    SDK defined as4096 */
    &attrNames);   /* Buff. Point.*/
  if (cCode < 0) {
    return(cCode);
  }
```

/ We must initialize all *input* buffers before we can use them. Note that buffers that pass info back from NetWare do not need t be initialized. Initialization also indicates the operation we'll be performing. */*

```
  cCode = NWDSInitBuf(
    dContext, /* Context  */
    DSV_READ,  /* Operation */
    attrNames);  /* buffer   */
  if (cCode < 0) {
    NWDSFreeBuf(attrNames);
    return(cCode);
  }
```

/ Here's where put insert what we're interested - the "profile" attribute of the user object. Note that you can fetch values for more than one attribute. Call NWDSPutAttrName for each attribute in which you're interested. */*

```
  cCode = NWDSPutAttrName(
    dContext,  /* context   */
    attrNames, /* in buffer */
    "Profile");  /* attribute name */
  if (cCode < 0) {
    NWDSFreeBuf(attrNames);
    return(cCode);
  }
```

/ The next buffer is where NetWare will return the results. Since it's passing info back, we don't need to initialize it. */*

```
cCode = NWDSAllocBuf(
  DEFAULT_MESSAGE_LEN, /* Buffer Size */
  &outBuf);      /* Buff. Point.*/
  if (cCode < 0) {
    NWDSFreeBuf(attrNames);
    return(cCode);
  }
```

/ The do loop fetches all the attribute names and associated values. If we had put in multiple attributes, this would cycle through all the attributes and their associated returned values. */*

```
do {
    cCode = NWDSRead(
      dContext,  /* Context */
      objectName,  /* Object name */
      DS_ATTRIBUTE_VALUES, /* Info. Type --
                        Return names and
                        values */
      FALSE,     /* All Attrib */
      attrNames, /* Attrib. names */
      &iterHandle,   /* Iter. Handle */
      outBuf);   /* Object info */

    if (cCode < 0) {
      NWDSFreeBuf(attrNames);
      NWDSFreeBuf(outBuf);
      return(1);
    }

    cCode = DisplayAttributes(dContext,
      outBuf);
    if (cCode < 0) {
      NWDSFreeBuf(attrNames);
      NWDSFreeBuf(outBuf);
      return(1);
    }

  } while(iterHandle != -1L);

  NWDSFreeBuf(attrNames);
  NWDSFreeBuf(outBuf);

  return(0);
}

NWDSCCODE DisplayAttributes(
  NWDSContextHandle dContext, pBuf_T buf)
{
  NWSYNTAX_ID  syntax;
  NWDSCCODE  cCode = 0;
  NWCOUNT  attrCount;
  NWCOUNT  valCount;
  char   attrName[MAX_DN_CHARS + 1];
  void   *attrVal;
  NWSIZEattrValSize;
```

/ Find out how many attributes we have. Our example program only has one, but we still have to check.*/*

```
  cCode = NWDSGetAttrCount(
    dContext,    /* Context */
    buf,    /* attr. buff */
    &attrCount); /* num of attr*/
  if (cCode < 0)
    return(cCode);
```

/ Pick out a attribute name from the returned buffer. This will also return the "syntax" value for the attribute. We'll need that next...*/*

```
  cCode = NWDSGetAttrName(
    dContext,  /* Context */
    buf,  /* attrib. buf */
    attrName,  /* attrib name */
    &valCount, /* attr. val. cnt */
    &syntax);  /* Syntax ID */
  if (cCode < 0)
    return(cCode);
```

/ Next, we find out how large the attribute's size is before we actually pull it from the buffer. This lets us alloc the required memory. */*

```
  cCode = NWDSComputeAttrValSize(
    dContext,  /* Context handle */
    buf,   /* Result Buffer */
    syntax,  /* Syntax ID */
    &attrValSize); /* Size of attrib.*/

  attrVal = (void *)malloc(attrValSize);
```

/ Pop out the attribute's value */*

```
  cCode = NWDSGetAttrVal(
    dContext,  /* Context */
    buf,   /* result buf */
    syntax,    /* syntax id */
    attrVal);  /* attr. val */
  if (cCode < 0)
    return(cCode);
```

/ Now that we know which object we need to pull the "login Script" attribute from, do it */*

```
  cCode = GetObjectScript(dContext,
    attrVal);

  free(attrVal);     /* clean up */

  return 0;

}
```

/ Compile and run script text using the Mac's default scripting environment – see Inside Macintosh: Interapplication Communication for more details */*

```
OSAError RunScript(Handle scriptText)
{
  OSAError err = noErr;    /* An err we
                           better check */
  OSAError ignoreErr = noErr;   /* An err
                           we can ignore */

  AEDesc scriptData;
  AEDesc resultData;
  AEDesc componentName;
  OSAID   scriptID;
  OSAID   resultID;

  short   i;

  scriptData.descriptorType = typeChar;
  scriptData.dataHandle = scriptText;

  gScriptingComponent =
    OpenDefaultComponent(kOSAComponentType,
    kOSAGenericScriptingComponentSubtype);

  err = OSAScriptingComponentName(
    gScriptingComponent, &componentName);

  if (err == noErr) {

    scriptID = kOSANullScript;
    err = OSACompile(gScriptingComponent,
      &scriptData, kOSAModeNull,
      &scriptID);
    ignoreErr = AEDisposeDesc(&scriptData);

    if (err == noErr) {

      err = OSAExecute(gScriptingComponent,
        scriptID, kOSANullScript,
        kOSAModeNull, &resultID);
      ignoreErr = OSADispose(
        gScriptingComponent, scriptID);

      if (err == noErr) {

        err = OSADisplay(
          gScriptingComponent, resultID,
          typeChar, kOSAModeNull,
          &resultData);

      }

      for (i = 0; i < kTimeOut; i++)
        SystemTask();

    }

  }

  return(err);
}
```