

3

The Basics of NBT Implementation

In theory, theory and practice are the same.
In practice, they're not.

— Unknown

Ready?

We have identified the three key parts of NBT: the Name Service, the Datagram Service, and the Session Service. This is enough to get us started. We will begin by coding up a simple Name Service Query, just to see what kind of trouble that gets us into.

Before we start, though, it's probably a good idea to check our tools.

Sniffer

You need one of these. If you have Windows systems available, see if you can get a copy of Microsoft's NetMon (**Net**work **Mon**itor). You will want the latest and most complete version. The advantage of NetMon is that Microsoft have included parsers for many of their protocols.

Another excellent choice is Ethereal, an Open Source protocol analyzer portable to most Unix-ish platforms and to Windows. It can create its own captures or read captures made by several other sniffer packages, including TCPDump and NetMon. Richard Sharpe and Tim Potter of the Samba Team have worked on NetBIOS and SMB packet parsers for Ethereal, which helps a big bunch.

✔ Language

There are a lot of programming languages out there. Samba is written in C, and jCIFS is in Java. The key factors when choosing a language for your implementation are:

- Good network coding capabilities.
- That warm fuzzy feeling you get when you code in a language you truly grok.

Meditate on that for a while. Bad karma in the coding environment will distract you from your purpose.

✔ Test Environment

If you do not have a couple of hubs, a router, various Windows boxes, and some Samba servers in your home, you may need to do your testing at the office. Netiquette and job security would suggest that you test after hours. (Um... actually, you probably shouldn't do any testing on a production network... and check office policy before you sniff.)

✔ Medication

An aromatic black tea, such as a good Earl Grey, is best. Try Lapsang Souchong to get through really difficult coding sessions. Those sweet, mass-produced, over-cafeinated soft drinks will disturb your focus.

Ready!

In this section, we will implement a broadcast NAME QUERY REQUEST. That is, B mode name resolution. This will allow us to introduce some of the basic concepts and establish a frame of reference. In other words, we have to start somewhere and this seems to be as good a place as any.

3.1 You Got the Name, Look Up the Number

The structure of an NBT name query is similar to that of a Domain Name System query. As RFC 1001, Section 11.1.1, explains:

The NBNS design attempts to align itself with the Domain Name System in a number of ways.

The goal of this attempted alignment was an eventual merger between the NBNS and the DNS system. The NBT authors even predicted dynamic DNS update. With Windows 2000, Microsoft did move CIFS naming services to Dynamic DNS, though the mechanism is not quite what was envisioned by the authors of the NBT RFCs.

3.1.1 *Encoding NetBIOS Names*

RFCs 1001 and 1002 reference RFC 883 when discussing domain name syntax rules. RFC 883 was later superseded by RFC 1035, but both give the same *preferred*¹ syntax for domain names:

```

<domain>      ::= <subdomain> | " "
<subdomain>   ::= <label> | <subdomain> "." <label>
<label>       ::= <letter> [ [ <ldh-str> ] <let-dig> ]
<ldh-str>     ::= <let-dig-hyp> | <let-dig-hyp> <ldh-str>
<let-dig-hyp> ::= <let-dig> | "-"
<let-dig>     ::= <letter> | <digit>
<letter>      ::= any one of the 52 alphabetic characters A through
                    Z in upper case and a through z in lower case
<digit>       ::= any one of the ten digits 0 through 9

```

This is the syntax that the NBT authors tried to match. Unfortunately, except for the 16-byte length restriction, there are few syntax rules for NetBIOS names. With a few notable exceptions just about any octet value may be used, so the NBT authors came up with a scheme to force NetBIOS names into compliance. Here's how it works:







- Names shorter than 16 bytes are padded on the right with spaces. Longer names are truncated.
- Each byte is divided into two nibbles (4 bits each, unsigned).² The result is a string of 32 integer values, each in the range 0..15.
- The ASCII value of the letter 'A' (65, or 0x41) is added to each nibble and the result is taken as a character. This creates a string of 32 characters, each in the range 'A'..'P'.

1. Note the use of the term "preferred." A close read of RFCs 883, 1034, 1035, and 2181 shows that the idea of using binary data in DNS records has been around for some time.

2. Some call this a "half-ASCII"-ed encoding scheme.

This is called *First Level Encoding*, and is described in RFC 1001, Section 14.1.

Using First Level Encoding, the name “Neko” would be converted as follows:

char		hex		split		+ 'A'		hex		result
N	=	0x4E		0x04	+	0x41	=	0x45	=	E
				0x0E	+	0x41	=	0x4F	=	O
e	=	0x65		0x06	+	0x41	=	0x47	=	G
				0x05	+	0x41	=	0x46	=	F
k	=	0x6B		0x06	+	0x41	=	0x47	=	G
				0x0B	+	0x41	=	0x4C	=	L
o	=	0x6F		0x06	+	0x41	=	0x47	=	G
				0x0F	+	0x41	=	0x50	=	P
' '	=	0x20		0x02	+	0x41	=	0x43	=	C
				0x00	+	0x41	=	0x41	=	A
' '	=	0x20		0x02	+	0x41	=	0x43	=	C
				0x00	+	0x41	=	0x41	=	A
										⋮

This results in the string:

```
EOGFGLGPCACACACACACACACACACACA
```

Lovely, isn't it?

...and our first bit of code is in Listing 3.1.

This function reads up to 16 characters from the input string name and converts each to the encoded format, stuffing the result into the target string `dst`. The space character (0x20) always converts to the two-character value CA so, if the source string is less than 16 bytes, we simply pad the target string with CACA. Note that the target character array must be at least 33 bytes long — one extra byte to account for the nul terminator.³

3. Such considerations are important when programming in C and its ilk.

Listing 3.1: First Level Encoding

```

#ifndef uchar
#define uchar unsigned char
#endif /* uchar */

uchar *L1_Encode( uchar *dst, const uchar *name )
{
    int i = 0;
    int j = 0;

    /* Encode the name. */
    while( ('\0' != name[i]) && (i < 16) )
    {
        dst[j++] = 'A' + ((name[i] & 0xF0) >> 4);
        dst[j++] = 'A' + (name[i++] & 0x0F);
    }

    /* Encode the padding bytes. */
    while( j < 32 )
    {
        dst[j++] = 'C';
        dst[j++] = 'A';
    }

    /* Terminate the string. */
    dst[32] = '\0';

    return( dst );
} /* L1_Encode */

```

**Typo Alert**

RFC 1001 provides an example of First Level Encoding in Section 14.1. The string “The NetBIOS name” is encoded as:

FEGHGFCAEOGFHEECEJEPFDCAHEGBGNF

Decoding this string, however, we get “Tge NetBIOS tame.” Perhaps it’s a secret message.

The correct encoding would be:

FEGIGFCAEOGFHEECEJEPFDCAQGBGNF

3.1.2 *Fully Qualified NBT Names*

Now that we've managed to convert the NetBIOS name into a DNS-aligned form, it is time to combine it with the NBT Scope ID. The result will be a fully-qualified NBT address, which we will call the *NBT name*. To be pedantic, when the RFCs talk about First Level Encoding, this fully qualified form is what they really mean.

As expected, the syntax of the Scope ID follows the DNS recommendations given in RFC 883 (and repeated in RFC 1035). That is, a Scope ID looks like a DNS name. So, if the Scope ID is `cat.org`, and the NetBIOS name is `Neko`, the resultant NBT name would be:

```
EOGFGLGPCACACACACACACACACACACACA.CAT.ORG
```

Imagine typing that into your web browser. This is why the RFC 1001/1002 scheme for merging the NBNS with the DNS never took hold.

3.1.3 *Second Level Encoding*

Now that we have an NBT name in a nice familiar format, it is time to convert it into something else.

DNS names (and, therefore, NBT names) are made up of labels separated by dots. Dividing the name above into its component labels gives us:

length	label
32	EOGFGLGPCACACACACACACACACACACACA
3	CAT
3	ORG
0	<nul>

The Second Level Encoded NBT name is a concatenation of the lengths and the labels, as in:

```
'\x20' + "EOGFGLGPCACACACACACACACACACACACA" + '\x03' + "CAT"
+ '\x03' + "ORG" + '\0'
```

The empty label at the end is important. It is a label of zero length, and it represents the root of the DNS (and NBT) namespace. That means that the final nul byte is *part of the encoded NBT name*, and not a mere terminator.

In practice, you can manipulate the encoded NBT name as if it were a nul-terminated string, but always keep in mind that it is really a series of length-delimited strings.⁴

Our second bit of code in Listing 3.2 will convert a NetBIOS name and Scope ID into a Second Level Encoded string.

Listing 3.2: Second Level Encoding

```
int L2_Encode( uchar *dst, const uchar *name, const uchar *scope )
{
    int lenpos;
    int i;
    int j;

    /* First Level Encode the NetBIOS name.
     * Prefix it with label length.
     * (dec 32 == 0x20)
     */
    if( NULL == L1_Encode( &dst[1], name ) )
        return( -1 );
    dst[0] = 0x20;
    lenpos = 33;

    /* Copy each scope label to dst,
     * adding the length byte as an afterthought.
     */
    if( '\0' != *scope )
    {
        do
        {
            for( i = 0, j = (lenpos + 1);
                ('.' != scope[i]) && ('\0' != scope[i]);
                i++, j++)
                dst[j] = scope[i];

            dst[lenpos] = (uchar)i;
            lenpos += i + 1;
            scope += i;
        } while( '.' == *(scope++) );
        dst[lenpos] = '\0';
    }
    return( lenpos + 1 );
} /* L2_Encode */
```

4. “Pedantic” is the politically correct way to say “anal retentive.”

Not the prettiest piece of code, but it does the job. We will run through the function quickly, just to familiarize you with the workings of this particular programmer's twisted little brain. If the code is fairly obvious to you, feel free to skip ahead to the next section.

```
if( NULL == L1_Encode( &dst[1], name ) )
    return( -1 );
dst[0] = 0x20;
lenpos = 33;
```

Call `L1_Encode()` to convert the NetBIOS name into its First Level Encoded form, then prefix the encoded name with a length byte. This gives us the first label of the encoded NBT name. Note that we check for a `NULL` return value. This is paranoia on the programmer's part since this version of `L1_Encode()` does not return `NULL`. (An improved version of `L1_Encode()` might return `NULL` if it detected an error.)

The variable `lenpos` is set to the offset at which the next length byte will be written. The `L1_Encode()` function has already placed a nul byte at this location so, if the scope string is empty, the NBT name is already completely encoded.

```
if( '\0' != *scope )
{
    do
    {
        :
    } while( '.' == *(scope++) );
    dst[lenpos] = '\0';
}
```

The processing of scope labels is contained within the `do...while` loop. If the scope is empty, we can skip this loop entirely. Note that the root label is added to the end of the target string, `dst`, following the scope labels.

```
for( i = 0, j = (lenpos + 1);
    ('.' != scope[i]) && ('\0' != scope[i]);
    i++, j++)
    dst[j] = scope[i];
```

Run through the current label, copying it to the destination string. The variable `i` keeps track of the length of the label. A dot or a nul will mark the end of the current label.


```
dst[lenpos] = (uchar)i;
lenpos     += i + 1;
scope      += i;
```

Write the length byte for the current label, and then move on to the next by advancing `lenpos`. The variable `scope` is advanced by the length of the current label, which should leave it pointing to the dot or nul that terminated the label. It will be advanced one more byte within the `while` clause at the end of the loop.

Hopefully that was a nice, short waste of time. As we progress, it will become necessary to move more quickly and provide less code and less analysis of the code. There is a lot of ground to cover.

3.1.4 Name Service Packet Headers

Once again, our attention is drawn to the ancient lore of RFC 883, which was written about four years ahead of RFC 1001/1002 and was eventually replaced by RFC 1035. The comings and goings of the RFCs are a study unto themselves.

NBT Name Service packets are an intentional rip-off of DNS Messages. New flag field values, operation codes, and return codes were added but the design was in keeping with the goal of eventually merging NBNS services into the DNS.

This, conceptually, is what a Name Service packet header looks like:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NAME_TRN_ID															
R	OPCODE				NM_FLAGS						RCODE				
QDCOUNT															
ANCOUNT															
NSCOUNT															
ARCOUNT															

Here is a description of the fields:

NAME_TRN_ID

A two-byte transaction identifier. Each time the NBT Name Service starts a new transaction it assigns an ID to it so that it can figure out which

responses go with which requests. An obvious way to handle this is to start with zero and increment each time one is used, allowing rollover at 0xFFFF.

For our purposes any number will do. So we will pick something semi-random for ourselves. How 'bout 1964?

R

This one-bit field indicates whether the packet is:

- 0 == a *request*, or
- 1 == a *response*.

Ours is a request. It initiates a transaction, so we will use 0.

OPCODE

Six operations are defined by the RFCs. These are:

- 0x0 == Query
- 0x5 == Name Registration
- 0x6 == Name Release
- 0x7 == WACK (Wait for Acknowledgement)
- 0x8 == Name Refresh
- 0x9 == Name Refresh (Alternate Opcode)

The 0x9 OpCode value is the result of a typo in RFC 1002. In Section 4.2.1.1 a value of 0x8 is listed, but Section 4.2.4 shows a value of 0x9. A sensible implementation will handle either, though 0x8 is the preferred value.

One more OpCode was added after the RFCs were published:

- 0xF == Multi-Homed Name Registration

Our immediate interest, of course, is with the Query operation — OpCode value 0x0.

NM_FLAGS

As the name suggests, this is a set of one-bit flags, as follows:

0	1	2	3	4	5	6
AA	TC	RD	RA	0	0	B

We will go into the details of this later on. For now, note that the B flag means “Broadcast” and that we are attempting to do a broadcast query, so we will want to turn this bit on. We will also set the RD flag. RD stands for “Recursion Desired”; for now, just take it on faith that this bit should be set. All others will be clear (zero).

RCODE

Return codes. This field is always 0x00 in request packets (those with an R value of zero). Each response packet type has its own set of possible RCODE values.

QDCOUNT

The number of names that follow in the query section. We will set this to 1 for our broadcast name query.

ANCOUNT

The number of answers in a POSITIVE NAME QUERY RESPONSE message. This field will be used in the replies we get in response to our broadcast query.

NSCOUNT, ARCOUNT

These are “Name Service Authority Count” and “Additional Record Count,” respectively. These can be ignored for now.

So, for a broadcast NAME QUERY REQUEST, our header will look like this:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1964															
0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
1															
0															
0															
0															

To make it easier to write the code for the above query, we will hand-convert the header into a string of bytes. We could do this in code (in fact,

that will be necessary for a real implementation), but dealing with such details at this point would be an unnecessary tangent. So...

```
unsigned char header[] =
{
    0x07, 0xAC, /* 1964 == 0x07AC.      */
    0x01, 0x10, /* 0 0000 0010001 0000 */
    0x00, 0x01, /* One name query.      */
    0x00, 0x00, /* Zero answers.        */
    0x00, 0x00, /* Zero authorities.    */
    0x00, 0x00 /* Zero additional.     */
};
```

3.1.5 *The Query Entry*

The query entries follow the header. A query entry consists of a Second Level Encoded NBT name followed by two additional fields: the `QUESTION_TYPE` and `QUESTION_CLASS`. Once again, this is taken directly from the DNS query packet.

Under NBT, the `QUESTION_TYPE` field is limited to two possible values, representing the two types of queries that are defined in the RFCs. These are:

NB == 0x0020

The `NAME QUERY REQUEST`, which we will use to perform our broadcast query.

NBSTAT == 0x0021

The `NODE STATUS REQUEST`, also known as an “Adapter Status” query. The latter is a reference to the original NetBIOS API command name.

Only one `QUESTION_CLASS` is defined for NBT, and that is the Internet Class: `0x0001`.

So, our completed `NAME QUERY REQUEST` packet will consist of:

- the NBT header, as given above,
- the Second Level encoded NBT name,
- the unsigned short values `0x0020` and `0x0001`.

3.1.6 *Some Trouble Ahead*

It would seem that it should now be easy to send a broadcast name query. Just put the pieces together and send them to UDP port 137 at the broadcast address. Yes that *should* be easy... except that we are now crossing the line between *theory* and *practice*, and that means trouble. Be brave.

Upper Case/lower case

RFCs 883 and 1035 state that DNS name lookups should be case-insensitive. That is, CAT.ORG is equivalent to cat.org and Cat.Org, etc. Case-insensitive comparison is not difficult, and First Level Encoding always produces a string of upper-case characters in the range ‘A’..‘P’, so we should have no trouble comparing

```
EOGFGLGPCACACACACACACACACACACACA.CAT.ORG
```

against

```
EOGFGLGPCACACACACACACACACACACACA.cat.org
```

...but what about the original NetBIOS name? The strings “Neko” and “NEKO” translate, respectively, to

```
EOGFGLGPCACACACACACACACACACACACA
```

and

```
EOEFELEPCACACACACACACACACACACACA
```

These strings do not match, and so we seem to have a problem. Are the two original names considered equivalent? If so, how should we handle them?

RFC 1001 and 1002 do not provide answers to these questions, so we need to look to other sources. Of course, the ultimate source for Truth and Wisdom is empirical information. That is, what actually happens on the wire? A little packet sniffing and a few simple tests will provide the answers we need. Here’s the plan:

1. Use lower-case or mixed-case names when configuring your test hosts.
2. Set up your sniffer to capture packets on UDP port 137.
3. Start or restart your test hosts.
4. After a few minutes, stop the capture.

If your sniffer can decode the NetBIOS names (Ethereal and NetMon can) then you will see that the NetBIOS names are all in upper case. This is normal behavior for NBT, even though it is not documented in the RFCs. Scope strings are also converted to upper case before on-the-wire use.

Here is another interesting test that you can perform if you have Windows and Samba systems (and/or others) on your network:

1. Modify the NBT Name Query code (in Listing 3.3 below) so that it converts the NetBIOS name to *lower case* rather than upper case. (That is, change `toupper` to `tolower`.)
2. Recompile.
3. Start your sniffer.
4. Use the code to send some queries. Try group names in particular.

In tests using Samba 2.0 and Windows, Samba servers respond to these lower-case queries while Windows systems do not. This suggests that Windows compares the encoded string, while Samba is decoding the string and performing a case-insensitive comparison on the result. One might argue that Samba's behavior is more robust, but comparing names without decoding them first is certainly faster.⁵

NetBIOS Name Syntax

We have specified a syntax for NBT names. So far, however, we have said little about the syntax for the original NetBIOS name. RFC 1001 says only that the name may not begin with an asterisk (*), because the asterisk is used for "wildcard" queries.

At the low level there are few real rules for forming NetBIOS names other than the length limit. Applications, however, often place restrictions on the names that users may choose. Windows, for example, will only allow a specific set of printable characters for workstation names, yet Microsoft's `nbtstat` program is much more accepting.

For the implementor, this is a bit of a problem. You want to be sure that your code can handle any bizarre name that reaches it over the network, yet help the user avoid choosing names that might cause another system to choke.

5. Samba's behavior may change if no reason can be found to compare the decoded names. Decoding costs a few cycles, which could be significant in an NBNS implementation.

A good rule of thumb is to warn users against choosing any character that is not legal in a “best practices” DNS label.

Padding Permutations

The space character (0x20) is the designated padding character for NetBIOS names, but there are a few exceptions. One of these is associated with wildcard queries. When sending a wildcard query, the name is padded with nul bytes rather than spaces, giving:

'*'	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Which translates to: CKAAAAAAAAAAAAAAAAAAAAAAAAAAAA.

Samba will respond to either space- or nul-padded wildcard queries, but Windows will only respond if the name is nul-padded.⁶ Once again, this indicates that Samba decodes NBT names before comparison, but Windows does not.

Microsoft has added a few other non-space-padded names as well. These are special case names, used with particular applications. Still, they demonstrate the need for flexibility in our encoding and decoding functions.

Label Length Limits

We did not bother to mention earlier that the label length bytes placed before each label during Second Level Encoding are not 8-bit values. The uppermost two bits are used as flags, leaving only 6 bits for the label length. Normally these flag bits are both zero (unset), so we can ignore them for now and (as with so many other little details) deal with them later on.

With only 6 bits, the length of each label is limited to 63 characters. The overall length of the Second Level Encoded string is further limited to 255 bytes. Our example code does not have any checks to ensure that the Scope ID has the correct syntax, though such tests would be required in any “real” implementation.

6. The lab in the basement is rather sparse, and not all versions of Microsoft Windows can be tested. The reported behavior was detected on those variants that were available. We leave it as an exercise for the reader to verify that the behavior is consistent. Please let us know of any contradictory results.

The Fine Print at the End

The RFCs do not say so, but the last byte of the NetBIOS name is reserved.

The practice probably goes back to the early days and IBM. The 16th byte of a NetBIOS name is used to designate the *purpose* of the name. This byte is known as the “suffix” (or sometimes the “type byte”), and it contains a value which indicates the type of the service that registered the name. Some example suffix values include:

```
0x00 == Workstation Service (aka Machine Name or Client Service)
0x03 == Messenger Service
0x20 == File Server Service
0x1B == Domain Master Browser
```

The care and feeding of suffix values is yet another topic to be covered in detail later on. A suffix value of 0x00 is fairly common, so we will use that in our broadcast query. Note that this changes the encoding of the NetBIOS name. Once again, using the name Neko:

instead of EOEFEELEPCACACACACACACACACACACAA,
you get EOEFEELEPCACACACACACACACACACACACAAA.



Shorthand Alert

When writing a NetBIOS name, the suffix value is often specified in hex, surrounded by angle brackets. So, the name NEKO with a suffix of 0x1D would be written:

```
NEKO<1D>
```

It is also fairly common to use the “#” character to indicate the suffix:

```
NEKO#1D
```

We will use the angle bracket notation where appropriate.

3.1.7 Finally! A Simple Broadcast Name Query

This next bit of code is full of shortcuts. The packet header is hard-coded, as are the QUESTION_TYPE and QUESTION_CLASS. No syntax checking is done on the scope string. Worst of all, the program sends the query but does not bother to listen for a reply. For that, we will use a sniffer.

Tools such as the nmblookup utility that comes with Samba, or Microsoft’s nbtstat program, could also be used to send a name query.

The goal, however, is to implement these tools on our own, and the next bit of code gives us a start.⁷

Listing 3.3: Simple Broadcast Name Query

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <ctype.h>

#define NBT_BCAST_ADDR "255.255.255.255"

#ifdef uchar
#define uchar unsigned char
#endif /* uchar */

uchar header[] =
{
    0x07, 0xAC, /* 1964 == 0x07AC. */
    0x01, 0x10, /* Binary 0 0000 0010001 0000 */
    0x00, 0x01, /* One name query. */
    0x00, 0x00, /* Zero answers. */
    0x00, 0x00, /* Zero authorities. */
    0x00, 0x00 /* Zero additional. */
};

uchar query_tail[] =
{
    0x00, 0x20,
    0x00, 0x01
};
```

7. The program in Listing 3.3 has been tested on Debian GNU/Linux and OpenBSD. You may have to fiddle a bit to get it to work on other platforms. Under older versions of NetBSD, OpenBSD, Miami for Amiga, and possibly other BSD-derived TCP stacks, messages sent to the limited broadcast address (255.255.255.255) may not actually be sent as Ethernet broadcasts. On these systems, it will be necessary to change the value of `NBT_BCAST_ADDR` to the directed broadcast address of the local subnet (the local subnet broadcast address). This bug has been fixed in both NetBSD and in OpenBSD. See the original NetBSD bug report (#7682) for more information.

```

uchar *L1_Encode( uchar      *dst,
                  const uchar *name,
                  const uchar pad,
                  const uchar sfx )
{
    int i = 0;
    int j = 0;
    int k = 0;

    while( ('\0' != name[i]) && (i < 15) )
    {
        k = toupper( name[i++] );
        dst[j++] = 'A' + ((k & 0xF0) >> 4);
        dst[j++] = 'A' + (k & 0x0F);
    }

    i = 'A' + ((pad & 0xF0) >> 4);
    k = 'A' + (pad & 0x0F);
    while( j < 30 )
    {
        dst[j++] = i;
        dst[j++] = k;
    }

    dst[30] = 'A' + ((sfx & 0xF0) >> 4);
    dst[31] = 'A' + (sfx & 0x0F);
    dst[32] = '\0';

    return( dst );
} /* L1_Encode */

int L2_Encode( uchar      *dst,
              const uchar *name,
              const uchar pad,
              const uchar sfx,
              const uchar *scope )
{
    int lenpos;
    int i;
    int j;

    if( NULL == L1_Encode( &dst[1], name, pad, sfx ) )
        return( -1 );
    dst[0] = 0x20;
    lenpos = 33;

```

```

if( '\0' != *scope )
{
    do
    {
        for( i = 0, j = (lenpos + 1);
            ('.' != scope[i]) && ('\0' != scope[i]);
            i++, j++)
            dst[j] = toupper( scope[i] );

        dst[lenpos] = (uchar)i;
        lenpos      += i + 1;
        scope        += i;
    } while( '.' == *(scope++) );
    dst[lenpos] = '\0';
}

return( lenpos + 1 );
} /* L2_Encode */

void Send_Nbtn_Bcast( uchar *msg, int msglen )
{
    int          s;
    int          true = 1;
    struct sockaddr_in sox;

    s = socket( PF_INET, SOCK_DGRAM, IPPROTO_UDP );
    if( s < 0 )
    {
        perror( "Socket()" );
        exit( 0 );
    }

    if( setsockopt( s, SOL_SOCKET, SO_BROADCAST,
                   &true, sizeof(int) ) < 0 )
    {
        perror( "Setsockopt()" );
        exit( 0 );
    }

    if( 0 == inet_aton( NBT_BCAST_ADDR, &(sox.sin_addr) ) )
    {
        printf( "Invalid IP address.\n" );
        exit( 0 );
    }
    sox.sin_family = AF_INET;
    sox.sin_port   = htons( 137 );

```

```

    if( sendto( s,
                (void *)msg,
                msglen,
                0,
                (struct sockaddr *)&sox,
                sizeof(struct sockaddr_in) ) < 0 )
    {
        perror( "Sendto()" );
        exit( 0 );
    }

    close( s );
} /* Send_Nbtn_Bcast */

int main( int argc, char *argv[] )
{
    uchar  bufr[512];
    int     len;
    int     total_len;
    uchar  *name;
    uchar  *scope;

    if( argc > 1 )
        name = (uchar *)argv[1];
    else
        exit( EXIT_FAILURE );

    if( argc > 2 )
        scope = (uchar *)argv[2];
    else
        scope = "";

    (void)memcpy( bufr, header, (total_len = sizeof(header)) );

    len = L2_Encode( &bufr[total_len], name, ' ', '\0', scope );
    if( len < 0 )
        return( EXIT_FAILURE );
    total_len += len;

    (void)memcpy( &bufr[total_len], query_tail, sizeof( query_tail ) );
    total_len += sizeof( query_tail );

    Send_Nbtn_Bcast( bufr, total_len );

    return( EXIT_SUCCESS );
} /* main */

```

The updated `L1_Encode()` function takes two new parameters: `pad` and `sfx`. These allow us to specify the padding character and the suffix, respectively. The `L2_Encode()` function also takes these additional parameters, so that it can pass them along to `L1_Encode()`, and both functions make use of `toupper()` to ensure that the NetBIOS name and Scope ID are in upper case.

The function `Send_Nbtn_Bcast()` does the job of transmitting a block of data via UDP. The destination is port UDP/137 at the universal broadcast address. The program mainline simply strings together the various pieces of the NBT query, taking the NetBIOS name and Scope ID from the command line.

Compile the code and give the executable the name `namequery`. The program takes one or two arguments. The first is the NetBIOS name, and the second is the Scope ID (the Scope ID is optional). For example, on a Unix system the command line (including the `$` prompt) might be:

```
$ namequery neko cat.org
```

Start your sniffer with the filter set to capture only packets sent to/from UDP port 137. If you are using TCPDump or Ethereal, the filter string is: `udp port 137`. Depending on your OS, you may need to have Root or Administrator privileges in order to run the sniffer.

Run `namequery` with the input shown above, and then stop the capture. You should get something like this:

```
+ Frame 1 (100 on wire, 100 captured)
+ Ethernet II
+ Internet Protocol
+ User Datagram Protocol
- NetBIOS Name Service
  Transaction ID: 0x07ac
+ Flags: 0x0110 (Name query)
  Questions: 1
  Answer RRs: 0
  Authority RRs: 0
  Additional RRs: 0
- Queries
  + NEKO               <00>.CAT.ORG: type NB, class inet
```

This example is copied from Ethereal output.

Compare the parsed output provided by the sniffer against the hard-coded information in the program. They should match up. Next, try a query

using a name on your own network and take a look at the response. If you use the name of a Workgroup or NT Domain, you may get responses from several systems.

Another way to get multiple replies is to use a wildcard query. If all NBT nodes on your local LAN use the same Scope ID, and if they are not P nodes, then they will all respond to the wildcard name. To try this, you must first change the call to `L2_Encode()` within `main()` so that it passes `'\0'` as the padding character. That is:

```
total_len += L2_Encode( &bufr[total_len], name, '\0', '\0', scope );
```

Then recompile and give the asterisk as the NetBIOS name:

```
$ namequery "**"
```

Try using other tools such as `nbtstat` in Windows or Samba's `nmblookup` to generate queries, and spend a bit of time looking at the results of these captures. You can also simply let the sniffer run for a while. If your network is active you will see all sorts of NetBIOS packets fly by (particularly if you are on a shared rather than a switched LAN).

3.2 Interlude

We now have method, madness, and a vague sense of the direction. We are ready to head out on the open code. Let us first take a moment to meditate on what we have covered so far. Start by considering this mental image...

Imagine a cold, rainy autumn day. Still thinking of summer, you have forgotten to wear a jacket. The chill of the rain runs through your entire body as you hurry along the street. You try to keep your neck dry by pulling up your thin sweater and hunching your shoulders. Down the road you spot a café. It looks warm and bright inside. You quicken your pace, then dash through the door as the drizzly rain becomes more enthusiastic and thunder rumbles in the distance.

The shop is cozy, but not too small. There are potted plants scattered about. Light jazz plays over well-hidden speakers. The clientele are trendy urban business types having quiet, serious discussions in pairs at small tables. Paintings by a local artist hang on the walls.

You step up to the counter. A young woman with a dozen earrings and short-cut hair smiles and asks you what you would like. A nice, hot cup of tea.

She reaches down behind the counter and grabs a large white mug. Then she opens a box and pulls out a tea bag that is at least three years old, drops it into the mug, and pours in hot water from the sink. “Three dollars” she says, still smiling.

If you are a coffee drinker, you probably don’t understand. Replace “*opens a box and pulls out a tea bag*” with “*opens a jar and scoops out one spoonful of freeze-dried instant*” and you will get the point. The point is that details matter. Certainly, an old tea bag in warm water will make a cup of tea... but not one worth drinking.⁸

Just so, our examples provide some working code but are far from satisfying. If we are going to write something truly enjoyable we need to dig into the details.

Let’s get to it.

8. With a few notable exceptions, this is the way tea is prepared in American cafés. Ick.