Using IBM's NetBios from Forth

Peter Knaggs

Department of Computing and Information Systems,
University of Paisley, High Street,
Paisley, Scotland. PA1 2BE

Abstract

A general overview of the IBM NetBios system is given and its Multi-Tasking abilities are discussed. A FORTH interface that exploits these is presented along with an example application program, which illustrates the integration of NetBios with FORTH's Multi-Tasker is described.

Introduction

The *Net*work *B*asic *I*nput/*O*utput *S*ystem (NetBios) is an "application program interface" [2] between an application task and a *L*ocal *A*rea *N*etwork (LAN) designed to provide a common communication capability between IBM PCs and compatibles. It has been implemented on a wide variety of physical networks including Ethernet, Token ring, Insertion ring, etc.

NetBios provides a communication link (or connection) between named entities using two main forms of communication, known as *sessions* and *datagrams*. Any application may add a name to the network. In a FORTH Multi-Tasking system it would be possible to provide two separate application tasks, each with its associated name, on the same host machine. The two tasks would then communicate with each other using the NetBios and neither task need know where the other is situated.

All requests to the NetBios are made using a *N*etwork *C*ontrol *B*lock (NCB) supplied by the application program. The NCB holds parameters for the network call and, on completion, contains status information.

Functions

The functions provided by NetBios can be broken down into five groups [2, 4]: Naming, Sessions, Datagrams, Broadcasting and General Housekeeping.

Naming

Each network card has its own unique physical name. To use this name in an application would be too restrictive as such an application would be forced to know with which physical system to communicate. NetBios provides some naming capabilities to allow applications to refer to logical, rather than physical, names thus allowing a network application to be independent of any physical machine.

The *Add Name* function will add a given logical name to the network, provided that the name is unique. This will provide a logical name for the physical system performing the add name function. Each physical system may have up to 255 logical names that could be used by different tasks or applications. It should be noted that while the NetBios provides for 255 logical names, in practise most network cards have a hardware limit of 16 names.

The function *Add Group Name* will add a group name to the network. A number of different physical systems can add the same group name, in which case they are considered to be a member of the same group. Thus a group name is a logical name that may refer to any number of physical machines. This facilitates communication to a selected group of machines, perhaps with special hardware to facilitate certain tasks. As with logical names each physical system may be in several different logical groups. Thus any given machine may have a number of unique logical names and a number of group names (a Group Name will use a logical name slot).

Remove is used to remove a name from the Network. If the name is an logical name, the name is completely removed. If it is a group name, the machine is removed from the group.

Sessions

A Session provides a one-to-one connection, analogous to a telephone call.

A Session is started by one application making a call to another. The called application must be listening for an incoming call. To call another application, the *Call* function is used. The *Listen* function is used to wait for an incoming call, while *Hangup* is used to disconnect the call. If you call a group name, only one member of the group will receive the call. This is known as making a virtual circuit [6].

Once the connection has been established the applications can exchange data (up to 64K at a time) with the guarantee that it will arrive. To exchange data, one application must use the *Transmit* function while the other is using a *Receive* function. If one side issues a transmit before the other has issued the corresponding receive, the data will be buffered until the receive is issued.

Datagrams

A datagram is a one-shot communication of up to 512 bytes.

If we can equate a session with a telephone call, then we can equate a Datagram with a letter in a postal system. Each letter (datagram) is delivered separately, thus it must carry the complete destination address. If the letter is lost the system does not time-out and automatically send a duplicate; error control is the user's responsibility. Finally, letters do not necessarily arrive in the order they are mailed [6].

With a Session a connection is first established and data is then transmitted along the connection. The connection remain open until the end of the session when it is terminated. With a Datagram a new connection is established, the data transmitted and the connection is terminated for each datagram.

A *Datagram Transmit* will send a datagram to a given name. The receiving name must be waiting to receive it otherwise it will be lost. When a datagram is sent to a logical name, only that name will receive it. However, if it is sent to a group name, all the members of that group will receive a copy.

A *Datagram Receive* will wait for a datagram to be received by a given name. A datagram transmitted from any name to the given name will be accepted. A datagram receive request must be pending to receive a datagram, any datagram sent when a datagram receive request is not pending will be lost.

Broadcasting

A Broadcast is a special form of datagram that is sent to all names. The *Broadcast Transmit* function will send a datagram to all names known to the network. The *Broadcast Receive* function is similar to the datagram receive function, except it will only receive a Broadcast message, thus a broadcast message will only be received by names that have a broadcast receive pending.

House keeping

There are four basic functions that are designed for the network manager to control the network system. The *Reset* function is used to totally reset the network card. *Network Status* will return the current status of the network card. A *Cancel* function is used to cancel a given command. Finally the *Un-Link* function disconnects from a remote disk server.

Invoking NetBios Functions

All NetBios functions are invoked in the same manner. The data required by the function is placed in the relevant fields of the NCB and the NetBios system call is invoked. This will take the NCB and post it into the NetBios for processing. The actual processing of the function is interrupt driven and will run concurrently with the application program.

NetBios has three different ways of returning back to the application program. The first is referred to as a *Wait* function, where NetBios will process the complete function before returning to the application.

The second is to post a *No-Wait* function. NetBios will add the function to its internal list of functions and return to the application directly. The application program must poll the "command complete" flag of the NCB to determine if the NetBios has completed the function.

The final method is to post a *No-Wait* function giving the address of an interrupt or callback routine. The NetBios will add the function request to its internal list and return to the application program. When the function has been completed, it will invoke the callback routine.

Multi-Tasking

In order to exploit the concurrent execution abilities of Forth and the NetBios, we use the "*No-Wait with callback*" invocation method. When a NetBios function is used, the invoking task will typically execute a `STOP` after making the NetBios call.

In the Forth/NetBios interface, a field has been added to the NCB to store the identity of the invoking task. The callback routine passed to the NetBios is always a "*wake task*" routine that extracts the task identity from the NCB and sets the task status to active, thus waking the task associated with the NetBios function.

More than one task can have a NetBios request pending. For example, one task may be waiting on a *Broadcast Receive* whilst another is waiting on a *Datagram Transmit*. Any one task may have several NetBios requests pending. For example, in the "Net-Chat" application, one of the tasks posts four *Datagram Receive* requests to ensure that no incoming datagrams are lost (see *Net-Chat* and *The "Net-Chat" Application*). When the task is made active it has to poll the NCBs of the pending commands in order to discover which of them has completed. The NetBios does not impose a limit on the number of requests pending, although a network card might, there is however a cost as each request *must* have a separate NCB.

Examples

In this section we provide the reader with two examples of how the Forth/ NetBios interface can be used.

Block Transfer

To transfer a block of data from one system to another, both systems must make themselves known to the network. This would be done by each of them creating an NCB. They would then add their logical names to the network.

```
System One                        System Two
NEWNCB NCB                        NEWNCB NCB
" PETER" NCB ADD-NAME             " JOHN" NCB ADD-NAME
```

Now PETER may call JOHN. The connection is made when Peter is calling John and John is listening for a call from Peter (or when John makes a call to Peter, although Peter must be listening for the call in this case).

```
" JOHN" NCB PHONE                 " PETER" NCB LISTEN
STOP                              STOP
```

PETER will now send a block of data over the network to JOHN.

```
9 BLOCK      ( Address of buffer )  10 BLOCK     ( Address of buffer )
1024         ( Number of bytes )    1024         ( Number of bytes )
NCB          ( NCB to use )         NCB          ( NCB to use )
TX STOP      ( Transmit )           RX STOP      ( Receive )
```

One of the systems must now disconnect. Our convention is that the caller is in charge of the connection and hence is responsible for the disconnection.

```
NCB HANGUP   ( Disconnect )
```

The STOPs are required to allow other tasks to continue executing and to synchronise communications.

Net-Chat

A simple example application program has been developed along the lines of the "Net-Chat" program by Glass [1]. This is a Citizen Band radio emulation, in that if anyone sends a message over "Net-Chat", it will be received by all other systems running the application.

The basic principle to a "Net-Chat" implementation is to have a group name of "NET-CHAT" and a logical name for each person on the system. The screen is divided into two sections with a small 5 line window provided for the Net-Chat display and a larger second window displaying the normal OPERATOR environment.

A task ("CHAT-TASK") will post four *Datagram Receive* requests on the group name NET-CHAT. When a datagram is sent to NET-CHAT, all the members in the group will receive a copy (including the sender). When receiving messages, CHAT-TASK will scan through the NCBs to discover which one was honoured. It will take the message buffer of the NCB, display it in the Net-Chat window and will use the NCB to post a new *Datagram Receive* request. If only a single *Datagram Receive* was posted, it would be possible to miss a datagram that arrives between the previous datagram being received and the *Datagram Receive* request being re-posted. Having multiple receive requests allow us to continue to receive messages while we are processing the first message, thus we should not miss any messages.

To send a message, the user must type the word CHAT. This will ask for a message to be sent. It will send the message buffer to the group name NET-CHAT.

The code and a more detailed description, is given in *The "Net-Chat" Application*.

Problems

As this system was originally intended for use with the Novix micro-processor system, it was developed using the *poly*Forth system. For various reasons [3] it was later ported to the Forth++ system. In this section we describe some of the problems that had to be overcome before this system became fully operational.

polyForth

The *poly*Forth system operated correctly when used in a network based environment. When we loaded the NetBios interface code, the system stopped operating altogether. The *poly*Forth code appeared to be correct while the interface code also appeared to be correct.

After some experimentation, we discovered that the problem only occurred when the *poly*Forth serial communications package was loaded. By forcing the system *not* to load this package, the problem was overcome. In order to continue with this project, it was necessary to convert this system for use with the Forth++ system. Thus, the real cause of the problem was never investigated.

Callbacks

The original version of this system used the *No-Wait and Poll* method of posting a NetBios function. This meant that when an application task had posted a NetBios function, it would enter a loop testing the command complete flag of the relevant NCB. As the task is actively waiting for the function to complete, it is scheduled for time by the multi-tasking scheduler.

The system was redeveloped to take advantage of the "*No-Wait with Callback*" ability of the NetBios. The system developed to utilise this facility is described in *Multi-Tasking*. The task posting a NetBios function is allowed to continue execution. Eventually the task will execute a `STOP`. When the NetBios function has been completed the NetBios will invoke the given callback code. This code will reset the associated task's status to *active* thereby making sure that the task will be executed.

This allows a task to post as many NetBios functions as it requires. It also allows the task to be removed from the scheduler's active tasks list. When one of the NetBios functions associated with the task has completed, it will add the task to the active task list, thus removing the responsibility of polling the command complete flag altogether.

Porting

The port from *poly*Forth to Forth++ was a very simple one with only one small problem. None of the code had to be changed with the exception of the two machine code words.

The *poly*Forth assembler system is designed to be as processor independent as possible, while the assembler provided with the Forth++ system is designed around the Intel 80x86 family of processors. The two machine code words had to be converted from the *poly*Forth assembler form into the Forth++ form. The function of the code was not altered in any way, nor was the machine code produced altered. The only alteration was to the source code in order to produce the same object code.

We also took this opportunity to exploit Forth++'s ability of holding 64 KBytes of strings to enhance the error messages and improve the error handling provided by the interface.

Comparison with C interface

When compiled, the NetBios interface shown in *Interface Code* forms a run-time library. The library comprises of 186 lines of Forth code and compiles to just 1.2 KBytes (when compiled under Forth++). A simple C interface [5] takes some 110 lines of code (1.8 KBytes when compiled) and 270 lines of compile time definitions to provide the same functionality as the `(net)` word. The C interface requires the application developer to have a full knowledge of the NetBios and the NCB. A full C library that provides the same functionality as the interface given in *Interface Code* requires some 115 KBytes (when compiled using Microsoft C).

As the C language does not directly cater for multi-tasking, such an interface has to use the *No-Wait* or *No-Wait and Poll* techniques for invoking a NetBios function. Using the *No-Wait and Poll* technique puts the onus on the application programmer to poll the command complete flag, thus does not provide the full abstraction one might hope for.

Interface Code

The following is an annotated source listing of the NetBios Interface provided for use with the Forth++ system.

Error Handler

Here we define the word "`(netable)`" to display an understandable network error message. It only displays the errors documented in the NetBios manual [2]. Any error code not defined in the manual will be displayed as "Unknown".

```
HEX

: (netable)
  CASES
    01 CASE ." Illegal Buffer Length"   END-CASE
    03 CASE ." Illegal Command"         END-CASE
    05 CASE ." Timed Out"               END-CASE
    06 CASE ." Message Incomplete"      END-CASE
    08 CASE ." Illegal Session Number"  END-CASE
    09 CASE ." No Resource Available"   END-CASE
    0A CASE ." Session Closed"          END-CASE
    0B CASE ." Command Cancelled"       END-CASE
    0D CASE ." Local Duplicate Name"    END-CASE
    0E CASE ." Name Table Full"         END-CASE
    0F CASE ." Name Not Registered"     END-CASE
    11 CASE ." Session Table Full"      END-CASE
    12 CASE ." Call Rejected"           END-CASE
    13 CASE ." Illegal Name Number"     END-CASE
    14 CASE ." Destination Not Found"   END-CASE
    15 CASE ." Name Not Found"          END-CASE
    16 CASE ." Remote Duplicate Name"   END-CASE
    17 CASE ." Name Deleted"            END-CASE
    18 CASE ." Session Aborted"         END-CASE
    21 CASE ." NetBios is busy"         END-CASE
    23 CASE ." Invalid LAN number"      END-CASE
    24 CASE ." Command not found"       END-CASE
    26 CASE ." Illegal Cancel Command"  END-CASE
    34 CASE ." Illegal Data Format"     END-CASE
    DROP
    ." Unknown"
  END-CASES
;
```

We now define the default action to be taken when a network error occurs. This is defined in the word `(neterror)`, it will abort the current operation and display an error message of the form:

Network Error code: 15 (Name Not Found)

Displaying the network return code and a text message relating to the code (if known). Note that the word `?CASE` takes a flag of the stack and executes the code between the `?CASE` and the `END-CASE` if the flag is true, otherwise it simply skips over the code.

```
: (neterror) ( n -- )
  CR ." Network Error code: " DUP . ASCII ( EMIT
  CASES
                FF  CASE ." Not Finished"           END-CASE
    DUP 50 FF WITHIN ?CASE ." Hardware Fault"    DROP END-CASE
    DUP 40 50 WITHIN ?CASE ." Unusual Condition" DROP END-CASE
    (netable)
  END-CASES
  ASCII ) EMIT CR ABORT
;

DECIMAL
```

Next we define the network error handling. This is provided by the word `NETERROR`, it takes the NetBios return code and invokes the word, the execution token of which is stored in the user variable `'NETERROR`, if there has been an error, otherwise it simply removes the return code. The defining word `USER*` is used to define a user variable at the next free slot in the user area.

```
USER* 'NETERROR

: NETERROR ( n -- )
  ?DUP IF  'NETERROR  @ EXECUTE  THEN
;
```

Finally we initialise the network error handler to be our default error handler.

```
' (neterror) 'NETERROR !
```

Network Control Block

In this part of the system we define the logical names for the fields of the network control block (NCB), these are the names as given in the manual. It should be noted that we are using the `@` symbol to indicate a segment and offset pair in accordance with the manual. The run-time action of these words is to return the address of the given field in the given NCB.

The word `pos` is a defining word, the size of the field (in bytes) is given on the stack, `pos` will then define a word, the action of which is to add the required byte offset to an address in order to give the address of the required field. We have added the `TASK@` field to hold the address of the invoking task. This is not part of the standard NCB structure but has been added to allow the callback routine to identify the associated task. Finally, the constant `ncb_size` is defined to hold the size of our NCB structure (in bytes).

```
: pos CREATE OVER C, + DOES> C@ + ;

0  \ Initial byte count

1 pos CMD        1 pos RETCODE   1 pos LSN         1 pos NUM
```

```
4 pos BUFFER@    2 pos LENGTH   16 pos CALLNAME   16 pos NAME
1 pos RTO        1 pos STO       4 pos POST@       1 pos LANA_NUM
1 pos CMD_CPLT  14 pos RESERVED  4 pos TASK@

CONSTANT ncb_size
```

Next we define some NCB control words. The first of these is `NEWNCB`, this will allocate `ncb_size` bytes of memory to act as an NCB. It also creates a word, the action of which is to place the address of this memory area onto the stack.

```
: NEWNCB ( -- )
  CREATE HERE ncb_size DUP ALLOT ERASE
;
```

The second control word is `TIME-OUT`, this is used to set the "Receive" and "Send" time-outs for a given NCB. The time-outs are given in increments of $\frac{1}{2}$ seconds. The system is initialised to no time-outs by default.

```
: TIME-OUT ( Receive-Time-Out Send-Time-Out NCB -- )
  DUP STO ROT SWAP C! RTO C!
;
```

The last of the NCB control words is `COPYNCB`. This is used to copy the data from one NCB to another.

```
: COPYNCB ( Source-NCB Destination-NCB -- )  ncb_size CMOVE ;
```

Assembler Interface

This is where we have developed the assembler code that interfaces between the Forth++ system and the NetBios.

First, we define a word `FIELD` that returns the byte offset of a named field in the NCB. As this word is being defined exclusively for use in code level definitions, we place its definition in the `ASSEMBLER` wordlist.

```
ASSEMBLER DEFINITIONS

: FIELD ' >BODY C@ ;

FORTH DEFINITIONS
```

We now define the callback code that is invoked by NetBios when it has completed a *No-Wait with Callback* operation. NetBios refers to such a callback as a *post* routine, thus the name for this assembler word `(post)`. On entry to this word, the `ES:BX` register pair are pointing to the start of the NCB that has completed, the status of all other registers are unknown, thus we can not make any assumptions about the state of the system (other than the value of `ES:BX`). The callback routine uses the address stored in the `TASK@` field of the NCB to discover which task is related to the NCB. It will then place a 1 in that task's `STATUS` variable, thereby adding that task to the scheduler active task list.

```
CREATE-INTERRUPT (post)
  DS PUSHSEG   BX PUSH   AX PUSH   ES AX MOV   AX DS MOV
  FIELD TASK@ 2+ ) BX@ AX MOV   AX PUSH
  FIELD TASK@ ) BX@ AX MOV   AX BX MOV
  DS POPSEG   1 # USER STATUS MOV
  AX POP   BX POP   DS POPSEG
IRET
END-CODE
```

This code is given as it is provided in the FORTH++ interface. We now give the code again in a commented Intel assembler format.

```
post: push ds          ; Save the registers
      push bx          ;   we are going to use
      push ax

      mov  ax,es       ; Copy ES to DS
      mov  ds,ax

      mov  ax,[bx+66]  ; Get the DS for the task
      push ax          ; Save it for later

      mov  ax,[bx+64]  ; Get the offset of the task

      mov  bx,ax       ; Save in BX
      pop  ds          ; Recover task's DS

      mov  [bx+0],#1   ; Set task's status to active

      pop  ax          ; Recover registers
      pop  bx
      pop  ds

      iret             ; Return from interrupt
```

The next word we define is `(net)`. This word will initialise the NCB with a given command (`CMD`), buffer (`BUFFER@`) and post routine (`POST@`). It will then invoke the NetBios interrupt asking the NetBios to perform the function indicated by the command number. The `POST@` value passed to this word is the 16 bit offset of the `(post)` routine. If this offset is 0, an address of `0000:0000` is placed in the `POST@` field. When the NetBios returns from the interrupt it provides a "return value" that is passed back to the calling word.

```
HEX

CODE (net) ( NCB Buffer Command 'Post -- Retcode )
  CX POP   AX POP   DX POP   DI POP
  AL FIELD CMD ) DI@ MOV   DS AX MOV
  AX FIELD BUFFER@ 2+ ) DI@ MOV   DX FIELD BUFFER@ ) DI@ MOV
  CX AX MOV   0 # AX = NOT        IF  CS AX MOV  THEN
  AX FIELD POST@ 2+ ) DI@ MOV     CX FIELD POST@ ) DI@ MOV
  DS AX MOV
  AX FIELD TASK@ 2+ ) DI@ MOV     BX FIELD TASK@ ) DI@ MOV
  ES PUSHSEG   BX PUSH   DS AX MOV   AX ES MOV   DI BX MOV
  5C INT   BX POP   ES POPSEG   0 # AH MOV   AX PUSH
NEXT
END-CODE

DECIMAL
```

Again, this code is given as it is provided in the FORTH++ interface. We now give a version of the same code, with comments, in Intel assembler format.

```
net:  pop   cx             ; CX = POST@ offset
      pop   ax             ; AX = NetBios command
      pop   dx            ; DX = BUFFER@ offset
      pop   di            ; DI = NCB offset

      mov   [di+00],al    ; Set NetBios command in the NCB

      mov   ax,ds
      mov   [di+06],ax    ; Set the BUFFER@ segment to the current DS
      mov   [di+04],dx    ; Set BUFFER@ to the given offset

      mov   ax,cx         ; Is POST@ offset zero?
      cmp   ax,#0
      jne   $1            ; Yes, then AX and CX = 0
      mov   ax,cs         ;  No, then set AX to current CS

$1:   mov   [di+46],ax    ; Set POST@ segment to CS (0000 if CX=0000)
      mov   [di+44],cx    ; Set POST@ offset to CX

      mov   ax,ds
      mov   [di+66],ax    ; Set TASK@ segment to current DS
      mov   [di+64],bx    ; Set TASK@ offset to task user area

      push  es            ; Save registers ES:BX
      push  bx
      mov   ax,ds
      mov   es,ax         ; ES:BX = NCB address
      mov   bx,di
      int   5Ch           ; Invoke NetBios interrupt

      pop   bx            ; Recover ES:BX
      pop   es

      mov   ah,#0         ; Clear top byte of "Return Value"
      push  ax            ; Return "Return value"

      NEXT                ; Re-enter inner interpreter
```

Low-Level interface

The next part of the interface defines the low-level Forth words that are used to interface with the assembler definitions.

The first of these words is +NET. It will post a NetBios function and wait for it to complete before returning. It will then process the "Return Value", checking it for errors.

```
: +NET ( Buffer NCB Command -- )
  ROT SWAP 0 (net) NETERROR
;
```

The second word being -NET which will post a network function to the NetBios system using the *No-Wait with Callback* variant of the command. The calling task will be placed in the scheduler's active list on completion of the function. However, the task is not removed from the active list by this word. This is left to the application.

```
: -NET ( Buffer NCB Command -- )
  128 OR ROT SWAP (post) (net) NETERROR
;
```

We now define the word COMPLETE to check the NCB command complete (CMD_CPLT) flag. It will return a TRUE when the function has completed. This word is provided so that an application may test which of several possible NetBios commands has been honoured (see *Multi-Tasking* and *Net-Chat* for a description and *Listening* for an example of its use).

```
: COMPLETE ( NCB -- f )
  CMD_CPLT C@ 255 = NOT
;
```

The final definition in this section is NERROR which is used in conjunction with the COMPLETE word. It will check the return code (RETCODE) of a given NCB returning the NetBios return code, if the function associated with the NCB has completed, otherwise it returns a -1.

```
: NERROR ( NCB -- n )
  DUP COMPLETE IF RETCODE C@ ELSE DROP -1 THEN
;
```

General Support

Here we define a number of words for the general administration of the network. Most of these commands would only be used by a supervisor or supervising software. These commands do not have *No-Wait* variants, thus they all wait for the NetBios command to complete before returning to the caller.

NET-RESET will *Reset* the network with the support for the given number of sessions and the given number of outstanding commands using the given NCB.

```
: NET-RESET ( #sessions #commands NCB -- )
  DUP >R NUM C! R@ LSN C! 0 R> 50 +NET
;
```

NET-CANCEL is used to *Cancel* a NetBios command. The NetBios command associated with NCB1 is cancelled (removed from the command-pending list). Due to the way that the NetBios system operates, it requires a second NCB to be used to issue the cancel command.

```
: NET-CANCEL ( NCB1 NCB2 -- )  53 +NET ;
```

The UNLINK word will disconnect the node from the “Remote Program Link”. This is only used when booting the system over a network.

```
: UNLINK ( NCB -- )       DUP 112 +NET ;
```

Finally the NET-STAT word returns the current status of the network to the given buffer (addr) of a given maximum size (len1 bytes). Returning the number of bytes (len2) of actual data received. This data is dependent on both the network hardware and the particular NetBios implementation.

```
: NET-STAT ( addr len1 NCB -- len2 )
  SWAP OVER LENGTH DUP >R ! DUP CALLNAME ASCII * SWAP C!
  51 +NET R> @
;
```

Naming Support

In this section we define the Forth words that will give the programmer access to the NetBios “Naming” functions.

Firstly, the word (name) is defined. This word takes a counted string (s) as a symbolic name. It will place the name in the given NCB's NAME field. This takes a fixed 16 character name, thus `(name)` also pads out the field with zeros. Having copied the name into the `NAME` field, it will then invoke the NetBios function given in `n` (either *Add Name* or *Add Group Name*). Notice that it uses `+NET` to invoke the function, thus the system will wait for the name to be added to the local name table before returning. This word forms the bases of both the `ADD-NAME` and `ADD-GROUP` words.

```
: (name) ( s NCB n -- )
  >R DUP NAME DUP 16 ERASE ROT COUNT ROT SWAP CMOVE
  0 SWAP R> +NET
;
```

The word `ADD-NAME` is used to add an logical name to the list of logical names for this node. It takes a counted string (`s`) and a `NCB`. It will add the name to the system, associating the name with the NCB. Any command sent out using that NCB will be issued under the given name. You must copy the NCB if you wish to post more than one (simultaneous) command under this name.

```
: ADD-NAME  ( s NCB -- )  48 (name) ;
```

The `ADD-GROUP` command works in much the same way as the `ADD-NAME` command with the one exception that the name added to the local node is a group name. Thus several different nodes may be known by the same name.

```
: ADD-GROUP ( s NCB -- )  54 (name) ;
```

The final word in this section is `REMOVE-NAME`. This will remove the name associated with the `NCB` from the local name table. If the NCB is associated with a group name, the node is removed from the group. The name is disassociated from the `NCB`, thus allowing the `NCB` to be associated with another name.

```
: REMOVE-NAME (   NCB -- )  0 SWAP 49 +NET ;
```

Session Support

In this section, we provide words that allow the application programmer to access the session handling facility of the NetBios.

Before we define the words that the application programmer is to use, we first define two words that perform most of the operations. These words are internal to the interface and are not meant to be used by the application programmer.

The first of these is `(cname)` which takes a counted string (`s`) and places it in the `CALLNAME` field of the given `NCB`. As with the `(name)` word, this also pads the field out to 16 characters by adding zeros. `(cname)` not only leaves the NCB address on the stack, it also places a 0 onto the stack to be used as a null buffer address. See the words `PHONE` and `LISTEN` to see how the word is used.

```
: (cname) ( s NCB -- 0 NCB )
  DUP CALLNAME DUP 16 ERASE ROT COUNT ROT SWAP CMOVE 0 SWAP
;
```

The second internal word is `(len)`. This will simply place the given buffer length (`len`) into the `LENGTH` field of the given `NCB` without removing the NCB address from the stack.

```
: (len) ( len NCB -- NCB )
  SWAP OVER LENGTH !
;
```

Having defined the two supporting words, we can now go on to define the words that the application programmer will use to gain access to the NetBios session capability. As we have already likened a session connection to a telephone connection, we use telephone-like words in our interface.

The word `PHONE` is used to establish a connection. This is similar to making a telephone *call* where you give the name of the recipient as a counted string (`s`). If the call is being made to a group name, only one member of the group will receive the call. The NetBios selects the group member, a one-to-one connection is made with one of the group members. The particular member is not known and is non-deterministic.

```
: PHONE     (         s NCB --      )   (cname) 16 -NET ;
```

The word `LISTEN` is similar to listening for a telephone call. You give the name of the node you are waiting to hear from as a counted string (`s`). However, you will only hear calls from that node, if another node is attempting to contact this name, the *listen* command will not register the call. When a call is detected, a connection (session) is established on both nodes.

There is a special name of "`*`" that will listen for a call from anyone. When a call is detected, the session (connection) is established and the name of the caller is placed in the `CALLNAME` field of the NCB.

```
: LISTEN    (         s NCB --      )   (cname) 17 -NET ;
```

The word `HANGUP` is used to disconnect the session. This is similar to someone hanging up the telephone to break the connection. We use the same convention as is used for telephones in that the caller is responsible for clearing the connection.

```
: HANGUP    (           NCB --      )    0 SWAP 18 -NET ;
```

We now have the words that will allow one to set up a connection but we are still unable to transfer data over this connection. The next two words provide this capability. The connection must be established prior to any attempt to transmit data.

To transmit data over the connection (to source the data) we use the `TX` word. This takes a buffer (`buff`) of `len` bytes (the maximum buffer size being 64 KBytes) and transmits it over the connection. As this is a session connection NetBios provides a guarantee that the data will arrive.

```
: TX        ( Buff Len NCB --      )      (len) 20 -NET ;
```

To sink (receive) the data the `RX` word is used. We give the system a buffer area (`buff`) with a maximum size of `len` bytes where it can place the data when it is received. When data has been received, the `LENGTH` field of the NCB will hold the actual number of bytes received. If the buffer is not large enough to hold all the data, the system will buffer the remaining data internally and report an error. Under these conditions an error code of 6 is placed in the `RETCODE` field of the NCB. It is the responsibility of the application programmer to detect and act on this condition by issuing another receive request.

```
: RX        ( Buff Len NCB --      )      (len) 21 -NET ;
```

The final word in this section is `CALL-STAT` which is used to obtain status information on the connection (session) associated with the given `NCB`. It is given a buffer (`buff`) of `len1` bytes into which it will place the current status. The `CALL-STAT` word will return the actual number of bytes used (`len2`) by the status information. The status information returned by this word is partly defined, however a large part of the data is dependent on the NetBios implementation.

```
: CALL-STAT ( Buff Len1 NCB -- Len2 )
  SWAP OVER LENGTH DUP >R ! 52 +NET R> @
;
```

Datagram Support

This is where we develop the Forth words that will give the application programmer access to the "Datagram" communication level provided by the NetBios. A datagram can be thought of as a packet of up to 512 bytes on the network. Unlike session communication, there is no built-in protocol associated with datagrams. The receiving node *must* be listening for an incoming datagram, otherwise it will not receive it. The NetBios provides no guarantee that the datagram will be delivered.

The first word we define in this section is `DTX`, the *Datagram Transmit* function. This will take an area of memory (`buff`) of `len` bytes in length (maximum size being 512 Bytes). This is sent, as a single unit, to the indicated node (whose name is given as the counted string `s`).

Notice how this word uses `(cname)` to copy the destination node name into the `CALLNAME` field of the NCB. The `NIP` is required to disregard the extra `0` that `(cname)` places on the stack. We use `(len)` to copy the byte length into the `LENGTH` field of the NCB. We can make the NetBios call with the `-NET` word.

```
: DTX ( Buff Len s NCB -- )
  (cname) NIP (len) 32 -NET
;
```

The *Datagram Receive* function is provided by the word `DRX`. This is given an area of memory to place the received data (`buff`) which is a maximum size of `len` bytes (maximum buffer size is 512 bytes). This word will wait for an incoming datagram addressed to the name associated with the NCB. On receiving a datagram, it will place as much data as it can in the buffer returning the actual number of bytes received in the `LENGTH` field of the NCB. Note that if the received datagram was too large for the receiving buffer, the buffer is filled, the remaining data is lost, and a return value of 6 is given (in the `RETCODE` field). The name of the sending node is placed in the `CALLNAME` field. See *Listening* for an example of using datagrams.

```
: DRX ( Buff Len NCB -- )  (len) 33 -NET ;
```

Broadcast Support

In this, the final part of the interface, we define the words that provide access to the NetBios "Broadcast" commands. A broadcast can be thought of as sending a datagram to everybody. If you are not listening for a broadcast, you will miss it. Like the datagram it will not be buffered for you. As with datagram support, we only need two words to provide broadcast support, one to transmit and one to receive.

The first of these words is `BTX`, providing the *Broadcast Transmit* function. This takes the address of the memory buffer (`buff`) of `len` bytes in length (maximum size of 512 bytes). The data is then transmitted to every node on the system.

```
: BTX ( Buff Len NCB -- )  (len) 34 -NET ;
```

The second word required to provide broadcast support is `BRX`, providing the *Broadcast Receive* function. As with `DRX`, the address of a receive buffer is given (`buff`) with a maximum length of `len` (maximum buffer size is 512 bytes). When the system receives a broadcast message, it will place up to `len` bytes in the buffer loosing any additional data. The `LENGTH` field holds the actual number of bytes received. The `CALLNAME` field will hold the name of the sending node. If more than one *Broadcast Receive* is posted, they will *all* receive the same message.

```
: BRX ( Buff Len NCB -- )  (len) 35 -NET ;
```

It should be noted that the words `(netable)`, `(neterror)`, `pos`, `FIELD`, `(post)`, `(net)`, `+NET`, `-NET`, `(name)`, `(cname)` and `(len)` are internal to the interface and should not be used when programming applications with this package.

The "Net-Chat" Application

The following is an annotated source listing of the "Net-Chat" example application as described in *Net-Chat*.

Memory Buffers

The first part of the application is to reserve the memory buffers that are going to be used. This section not only reserves the memory but also defines words that allow easy access to this memory.

We are going to require five NCBs and buffers. We first reserve the space for the five NCBs (one outgoing, four incoming). The number of bytes to reserve is calculated by multiplying the number of bytes required for an NCB (`ncb_size`) by five. We then initialise this memory to zeros using the `ERASE` word.

```
CREATE ncbs   ncb_size 5 * ALLOT   ncbs ncb_size 5 * ERASE
```

Thus the word `ncbs` will return the start address of a block of memory large enough to hold five NCBs. We now define a word `NCB` that take an NCB number and returns the address of the indicated NCB from our table.

```
: NCB  ( n -- NCB )        ncb_size * ncbs + ;
```

Now we do the same for the data buffers. This time the buffers are 60 bytes long and is given the name `buff`, while the accessing word is called `BUFF`.

```
CREATE buff          60 5 * ALLOT    buff  60 5 *  ERASE

: BUFF ( n -- buff )       60 * buff +        ;
```

We now define the word `name` that takes an NCB number and initialises the stack ready for a NetBios call to the *Datagram Receive* function, placing the corresponding buffer address (`buff`), the maximum size of the buffer (60) and the indicated NCB (`NCB`) on the stack.

```
: name ( n -- buff 60 NCB )
  DUP BUFF SWAP NCB 60 SWAP
;
```

Listening

In this section we define the "Listening" part of the application. This code will post four *Datagram Receives* to the NetBios and wait for one of them to be honoured. It will then display the name of the sender and a one line message.

The first item to define is the actor that is going to execute the code (`CHAT-TASK`). The actor is defined now so as to indicate that all the code that follows (upto the `CONSTRUCT` word) will be performed by the actor concurrently with the main system.

```
ACTOR CHAT-TASK
```

The first word we define in the section is `NET-LISTEN` which simply posts four *Datagram Receive* functions which will operate in unison. It should be noted that NCB 0 has been reserved for outgoing messages.

```
: NET-LISTEN
  5 1 DO
      I name DRX
  LOOP
;
```

When one of these *Datagram Receive* functions has been honoured, the system will execute the `NET-DISP` word. This will scan through the NCBs to discover which of them has been honoured. It will then display the name of the sender (taking it from the `CALLNAME` field) and the associated message. Finally it re-posts the *Datagram Receive* command.

```
: NET-DISP
  5 1 DO                             \ Scan through the incoming NCBs
    I NCB COMPLETE                   \ Has the command been honoured ?
      IF
         I NCB CR
         CALLNAME 16 0 DO            \ Display the CALLNAME filed
           DUP C@ ?DUP 0= IF LEAVE THEN EMIT 1+
         LOOP DROP
         ." : "                      \ Display a name separator
         I BUFF I NCB LENGTH @ TYPE  \ Display the message
         I name DRX                  \ Re-post the DRX
      THEN
  LOOP
;
```

The output from `NET-DISP` will be displayed in a small window at the top of the screen. The following line defines the window to start at the top left of the screen, being 78 characters wide and 5 lines high. The `WITH-BORDER` indicates that the window will have a line boarder displayed around it. Finally the window will be called `NET-WIN`.

```
1 1 78 5 WITH-BORDER CREATE-WINDOW NET-WIN
```

The last word to be defined in this section is `NET-GO`. This is the word that the `CHAT-TASK` will be asked to perform (by the `GO` word). It initialises the window and posts the initial four *Datagram Receive* requests. It then enters into an infinite loop waiting for one (or more) of the requests to be honoured when it will call the `NET-DISP` word to display the message and re-post the receive request.

```
: NET-GO
  NET-WIN <WIN                       \ Open the window.
    *WCLEAR                          \ Clear it
    *TITLE"  Net Chat "              \ Give it a title

    NET-LISTEN                       \ Post initial four DRX commands
    BEGIN
      STOP                           \ Wait for one to be honoured
      NET-DISP                       \ Display the message & re-post
    AGAIN
  WIN>;
;
```

The final act in this section is to indicate the completion of the code that is to be executed by the `CHAT-TASK` actor. This also completes the definition of the actor. Any words defined from this point on would not be accessible to the `CHAT-TASK` actor.

```
CHAT-TASK CONSTRUCT
```

Sending

In this section we define the "Sending" part of the application. In reality this consists of one definition. The word `CHAT` will ask the user to type in a one line message. It will then send the message as a datagram to the group name "`NET-CHAT`", thus any node with a *Datagram Receive* posted on the group name `NET-CHAT` will receive a copy of the message (including the sending node).

Firstly, the word locates the outgoing message buffer (buffer 0). It then erases the buffer making sure no other message is stored there. It now displays a message asking the user to input the message they wish to transmit. The message is read directly into the buffer with a maximum of 60 characters in length:

78	Characters in the display line
-16	Maximum characters in user name
-2	Name/Message separator (": ")
60	Total allowable size of message

The number of characters actually typed is taken as the size of the buffer. The buffer is sent to the group name `NET-CHAT` via the outgoing NCB (NCB 0). Finally, the word waits for the *Datagram Transmit* function to complete before returning to the user.

```
: CHAT
  0 BUFF                                    \ Find outgoing buffer
  DUP 60 ERASE                              \ Erase buffer
  CR ." Message: "                          \ Ask for the message
  DUP 60 EXPECT                             \ Read in the message
  SPAN @  " NET-CHAT"  0 NCB  DTX           \ Send the message
  STOP                                      \ Wait for NetBios to complete
;
```

Initialisation

In this part of the application, we provide the initialisation of the system. The word `GO` initialises the system for use with the "Net-Chat" application as outlined in *Net-Chat*.

The first part of the initialisation is to define a word that is going to become the network error handler for the application. This is a very simple word that simply ignores any errors. This definition is required so that the `INIT-CHAT` word can examine the return code and take appropriate action. (The default action will cause the system to abort on an error.)

```
: NO-ERROR  DROP ;
```

The next part of the initialisation process is coded into the word `INIT-CHAT`. This initialises the network handling side of the system. Firstly, it replaces the standard error handling with our error handling system (`NO-ERROR`). It will then ask the user to type in a unique name that it will use to identify the user to the other uses of the system. It attempts to add the name to the network (*Add Name*). If an error occurs a message is displayed and the user is asked to supply an alternative name.

When the logical name has been established (on the outgoing NCB, NCB 0), the error handler is reset back to the default. The `NO-ERROR` handler is only used to allow the word to extract the error code and ask for another name if necessary.

The group name NET-CHAT is added to the network (on NCB 1). The information placed in the NCB by the *Add Group Name* function is copied to the remaining incoming NCBs (2, 3 and 4).

```
: INIT-CHAT
  'NETERROR @                           \ Save the default error handler
  ['] NO-ERROR  'NETERROR !             \ Reset the error handler

  BEGIN
    CR ." Enter your name: "            \ Ask for a name
    0 BUFF DUP 1+ 16 EXPECT             \ Read the name (max 16 chars)
    SPAN @ SWAP C!                      \ Make buff a counted string
    0 BUFF 0 NCB ADD-NAME               \ Add name to Network
    0 NCB NERROR
  WHILE                                 \ While error in Add-Name
    CR ." Sorry, someone else is already using that name, try another."
  REPEAT                                \ Repeat input sequence

  'NETERROR !                           \ Reset error handler to default

  " NET-CHAT"  1 NCB  ADD-GROUP         \ Add the group name
  1 NCB DUP DUP
  2 NCB COPYNCB                         \ Copy the NCB data to NCB 2
  3 NCB COPYNCB                         \ '' NCB 3
  4 NCB COPYNCB                         \ '' NCB 4
;
```

The window for use by the OPERATOR actor is now defined to be 15 lines of 78 characters starting at line 8, complete with a line boarder.

```
1 8 78 15 WITH-BORDER CREATE-WINDOW OP-WIN
```

Finally, the word GO is defined. This is the word that the user will type to initialise the "Net-Chat" application.

The first action of GO is to call the INIT-CHAT word. Thus it asks for an logical name and initialise the NCBs. GO will then clear the screen (CLEAR) and turn the hardware cursor off (HWC-OFF) ready for the windowing environment. It will then redirect the OPERATOR output to the OP-WIN window (<WIN). Finally, the actor CHAT-TASK is sent the message (SEND") to initialise its window and listen for and display incoming messages (NET-GO).

```
: GO
  INIT-CHAT                             \ Initialise the Network
  CLEAR                                 \ Clear the screen
  HWC-OFF                               \ Turn the hardware cursor off
  OP-WIN <WIN                           \ Redirect output to OP-WIN window
  *WCLEAR                               \ Clear the window
  *TITLE"  Operator "                   \ Title the window
  CHAT-TASK SEND" NET-GO "              \ Set the CHAT-TASK listening
;
```

Close Down

In this, the final section of the application, we provide the code that will close down the application. All applications should provide a graceful close down, especially when they are using the services of some kind of server such as the NetBios.

There are a number of things we need to do to close down: stop the CHATTASK actor; remove the unique name from the system; cancel any outstanding commands; resign from the NET-CHAT group; tidy up the screen. The order in which these events occur is quite important. All of this can be accomplished in the one Forth word, CLOSE-CHAT. This is the word that the user will type when they wish to close or leave "Net-Chat".

Our first task is to force the CHAT-TASK actor to stop processing. This we do by forcing it to accept a new task (via the MUST SEND" operation). We ask CHAT-TASK to close its window (WIN>) and then to stop processing until further notice (HALT). Having stopped CHAT-TASK from receiving any messages, we are now able to alter the status of the network. We first remove the unique name from the name table (REMOVE-NAME). This provides us with a free NCB which we use to cancel the *Datagram Receive* requests that CHAT-TASK would have posted (NET-CANCEL). Notice how any task can cancel these requests as the NetBios is unaware of our tasking mechanism, thus does not consider a NetBios request to be owned by any particular task.

We are no longer able to send a message as we do not have a unique name. We are no longer able to see messages as CHAT-TASK is not running. We are no longer listening for messages sent to the NET-CHAT group as we have just cancelled all such requests. Thus we are now in a position to be able to resign our membership of the NET-CHAT group (REMOVE-NAME). Finally, we close the operations window (WIN>) and re-establish the cursor (HWC-ON).

```
: CLOSE-CHAT
  CHAT-TASK MUST SEND" WIN> HALT"      \ Close NET-WIN and stop task

  0 NCB REMOVE-NAME                    \ Remove outgoing unique name

  1 NCB  0 NCB  NET-CANCEL             \ Cancel the DRX commands
  2 NCB  0 NCB  NET-CANCEL
  3 NCB  0 NCB  NET-CANCEL
  4 NCB  0 NCB  NET-CANCEL

  1 NCB REMOVE-NAME                    \ Remove the group name

  WIN>                                 \ Close OP-WIN
  HWC-ON                               \ Turn hardware cursor on
;
```

If the user wanted to restart the application, he would simply type GO and he would be back in the application.

Acknowledgments

The work presented in this paper was preformed as a part of a research project supported by the UK Science and Engineering Research Council with a CASE award (number B/88502911) and Computer Solutions Ltd [3]. I would like to thank Andrew Haley of Computer Solutions Ltd. for giving me the idea and Bill Stoddart, my research supervisor, for his guidance and for simply putting up with me.

References

[1] B. Glass. Understanding NetBios. *Byte*, pages 301–306, January 1989.

[2] IBM Corporation. *NetBios Application Development Guide*, 1987.

[3] Peter J. Knaggs. *Practical and Theoretical Aspects of Forth Software Development.* PhD. Thesis, School of Computing and Mathematics, University of Teesside, 1994.

[4] Nine Tiles. *SimpleNetBIOS Reference Guide*, 1988.

[5] W. D. Schwaderer. *C Programmer's Guide to NetBIOS.* Howard W. Sams & Company, 1988.

[6] Andrew S Tenenbaum. *Computer Networks.* Prentice Hall, second edition, 1988.