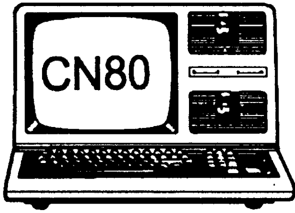


Computer News 80

PO Box 680, Casper, WY 82602

NewDos86

CUSTOM BASIC

for

NEWDOS80 USERS

by

WARWICK SANDS

(c) 1990

PUBLISHED AND DISTRIBUTED BY

COMPUTER NEWS 80
PO BOX 680
CASPER, WY 82602
U.S.A.

FOR SUPPORT CONTACT THE AUTHOR

WARWICK SANDS
139 SHAILER ROAD
SHAILER PARK
QUEENSLAND 4128
AUSTRALIA

PHONE: 7-801-2715

(061 COUNTRY CODE)

WARNING

NewDos86 can automatically change the PDRIVE settings to allow reading of 40 track diskettes in 80 track drives. So be careful not to write to any 40 track diskettes while they are in 80 track drives.

If you should write to a 40 track diskette in an 80 track drive, don't panic! You should always be able to read the diskette in the 80 track drive, even if it is no longer readable in 40 track drives.

If you should experience difficulties reading a 40 track diskette in a 40 track drive, try reading it in an 80 track drive. If you can read it in the 80 track drive, you can then copy the information back to a new diskette in your 40 track drive.

One final point. Some of the newer 80 track drives can write to 40 track diskettes successfully. So it might pay to try a few experiments with 40 track diskettes in your 80 track drives. Needless to say, don't experiment with disks that contain valuable data.

===

DIFFERENCES IN NEWDOS90 TO NEWDOS86

Why the name change to NewDos90? This is primarily because some changes have been made to the SYSTEM file usage. PDRIVE and SYSTEM, which use to occupy their individual /SYS files, now share a single /SYS file. This frees up a directory entry which is now used for a shell program.

This does mean that the DOS is now bigger and this may cause problems for those users who are still running single density systems. For this reason I will still be supporting NewDos86 to a limited extent. Where possible future utility files will try to support both DOS's. This is why you will see that some of the utilities are for NewDos86, while others are for NewDos90. All utilities for NewDos86 will run as designed under NewDos90, but the converse may not be true.

There have been some changes to the DOS and your manual may need updating.

New commands added:-
@ is an abbreviation for MDRET

* invokes the shell. A drive number may optionally follow the '*'. If you don't want to use the supplied shell, copy your preferred shell to SYS16/SYS. If you don't want any shell, issue the following command:-

```
DUMP sys16/sys 4CFFH,4CFFH,402DH
```

This creates a dummy file which passes control straight back to DOS.

SYS16/SYS is loaded if <SHIFT> is pressed on return to DOS ready or if bit 2 of byte 428CH (436CH model 1) is set. The <clear> key aborts the load. The shell program sets this bit to force an automatic reload of SYS16/SYS after executing DOS commands. See SHELL/TXT on ND90PD#2 for more details.

HELP filespec] invokes the helpdriver. The optional filespec is the file to view.

<SHIFT JKL> now causes a CHR$(12) (a formfeed) to be sent to the printer.

DOS now supports a POKE command (this is legal under Mini-DOS). Syntax is:-

```
POKE,[W]  exp1,exp2
```

The optional W indicates a WORD poke, while exp1 is the memory location poked. Exp1 and Exp2 can be anything that is legal in the ? command.

The ? command has been changed slightly. Opening and closing brackets are now permissible. NOTE that the Peek address must now be surrounded by brackets.

Examples:-

```
? p(5200H)                displays contents of location 5200H
? pw(5200H)               displays WORD stored at 5200H

POKE W 5200H+1,12         Stores 12 at 5201H, and 00H at 5202H
POKE 5200H,p(5200H)!2     Sets bit 1 of 5200H
POKE 5200H,p(5200H)&253   Resets bit 1 of 5200H
```

DOS now ignores leading and trailing whitespace when executing commands. If the first non-blank character of a command is an '=', then the DOS editor is invoked before the command is executed. This can be very handy within /JCL files.

Lowercase is now preserved when surrounded by double quotes. If you wish to strip the quotes from the command line. (E.g. while using forms to print lower case text), put a double quote as the first character on the line. If you then desire a double quote, use two of them. An example

PROT,0,RUF,NAME="Shell"
will cause the disk name to be "Shell", including the quotes.
"PROT,0,RUF,NAME="Shell
will cause the disk name to be Shell without the quotes.
"PROT,0,RUF,NAME=""Shell"!
will cause the disk name to be "Shell", including the quotes.

You can change the default quote character, for the current command only, to any of the following characters !"#$%&'() by having it as the first character in the command.

Leading and trailing white-space characters are stripped from the command line before it is executed and white-space characters within the line are converted to spaces. What is white space? Any character less than an ! mark. So down arrows, spaces etc are removed. Obviously <enter> still terminates the line.

If the first character of the command is an '=' then the '=' is replaced with a blank and the DOS edit routine is invoked. This is particularly useful inside JCL files when some parameters may need to be edited at run time.

Multiple commands are now allowed on the DOS command line. Separate DOS commands with a ~ (tilde). As each command is executed, it is removed from the DOS buffer. If an error occurs and DOS edit is invoked, the remaining commands can be edited. Since the line is stored in the DOS buffer, if Mini-DOS is entered then the remaining commands will be either be overwritten or executed.

===

```
 -------------------------------
|         ##############         |
|         ## NEWDOS86 ##         |
|         ##############         |
|                                |
| A FULLY INTEGRATED ENHANCEMENT |
| ============================== |
|    PACKAGE FOR NEWDOS8Ø V2     |
|    ======================      |
|                                |
| ############################## |
| ##       AND INTRODUCING    ## |
| ##        CUSTOM  BASIC     ## |
| ##  A POWERFUL ENHANCEMENT  ## |
| ##     TO  NEWDOS8Ø BASIC   ## |
| ############################## |
 -------------------------------
```

&%, &!, &H, &D, &O, &B, * SUFFIX, ADDED BASIC FUNCTIONS, ADDED DOS COMMANDS, AND$, AUTOMATIC DISK FORMAT DETERMINATION, BASE CONVERSIONS, BEEP, BIT TESTING, BOOT, CALCULATIONS (DOS), CALL USING, CAPTURE, CASE...ENDCASE, CLEAR *, CONFIGURATIONS (EXOTIC), CONTROL KEY DRIVER, CONTROL KEY (ELE), CONVERT BASIC PROGRAMS, COPY, CMD"F=POPx, DATE STAMPING (FILES), DAY, DEBUG (HI-MEM), DEC, DEF FN, DEF USR, DI-DU, DIR, #DO, DO...UNTIL, DOS KEY FUNCTIONS, #DRAW, EDIT DIRECT COMMANDS (BASIC), EDIT DIRECT COMMANDS (DOS), ENDKEY, ENHANCED LINE EDITOR, ENTERING BASIC (OPTIONS), ERROR (ERR$), EXITING BASIC (SIMPLER), FCB, FIELD @, FILL, FIX THE GAT TABLE (DIRECTORY REPAIR), FN, FORMAT, FORMAT (OTHER DOS'S), FORMS, FREE, FUNCTION KEYS (4/4P), GOSUB, #GOTO, HIMEM, INC, INPUT @, INPUT TO USING, INSTR, INSTR(!, KEYBOARD DRIVER, KEYBOARD TYPE AHEAD (KTA), KEYVAL, KTA (BASIC), $LET, LINELET, LOCATE, LOGICAL STRING FUNCTIONS, LOGON, LPRINT, LSET, LSTRIP, MASKING PARTSPECS, MID$=, MOD, #MOV, NEWDATE, OLDIR, OR$, PARTSPECS, PDRIVE, #PLOT, PRINT @, PRINTER DEFAULTS, PSEUDO SYSTEM FILES, PURGE, RENUMBER, #RESTORE, #REV, ROT$, RSET, RSTRIP, SETKEY, SETSYS, SHORTHAND BASIC COMMANDS, SINGLE KEY ENTRY, SORTDIR, SPOOLER, STRING SPACE SAVER, SUM, #SWP, $SWP, SYSTEM PARAMETERS (ADDED), TIME, TRSDOS PARSER, UNKILL, UPCASE$, USR, USR ROUTINES (RESERVE SPACE UNDER BASIC), VAR[L], VID (4/4P), WAIT, WHILE...WEND, WPEEK, WPOKE, XOR, XOR$

==

*** NOTICE ***
**** LIMITED WARRANTY ****

Although much care has gone into the production of both the program and the documentation, within the extent permitted by the law the author shall have no liability or responsibility to the purchaser or any other person, company or entity with respect to any liability, loss, or damage caused by or alleged to have been caused by this product, including, but not limited to, any interruption of service, loss of business and anticipatory profits or consequential damages resulting from the operation or misuse of this product.

Should the disks be unusable the author will replace the product on return of the disk in original condition. Except for this replacement policy the sale or subsequent use of the program is without warranty or liability.

This product is copyrighted with all rights reserved. The distribution and use of this package is intended for the personal use of the purchaser. The software is not copy-protected in any way. You can make as many copies for your own personal use as you like. It is not a shareware product.

The following trademarks may have been referenced in the Manual:- Newdos8Ø (Apparat Inc), Trsdos (Tandy Corp.), Ldos (Logical Systems Inc), Edas, Fed (Misosys).

There may be other copyrighted names mentioned in the manual -- no breach of copyright is intended by the use of these names. Since many of the owners of these names are now defunct, we have not conducted an exhaustive search as to which ones are still active. However, as most of these names seem to be part of the TRS-8Ø users vocabulary, they have been used in this sense.

****

Having got the legalese out of the way allow me to make a few comments. I realise that software piracy is a fact of life. To deny that it goes on is to play the ostrich. Shareware has tried to overcome this problem, but it appears that this concept of software distribution has to, a great extent, failed.

If you have received a copy of this package from a friend, please think seriously of becoming a legitimate owner of NewDos86. 'Why should I bother?' you ask. Firstly it is the honest thing to do and secondly it could well be that further program enhancements, patches and utility programs may well have been written since you received your copy.

The manual lists User Groups that distribute NewDos86. Contact one of these for the current price. The package is not expensive. Your support will help me to continue to support our TRS-8Ø machines and supply new programs and update current ones.

NEWDOS86 FEATURES ADDED TO ORIGINAL NEWDOS8Ø:

DISKETTE STORAGE HANDLING:

1) AUTOMATIC DISK FORMAT DETERMINATION (shields you from PDRIVEs as much as possible)
2) COPY & FORMAT greatly enhanced
4) DATE STAMPING of files
3) DIR (& PURGE) greatly enhanced with partspecs and masking etc
5) FREE now works for most disk (dos) types and shows granule map -- supports Double-Sided 4Ø & 8Ø-trackers
6) PURGE (see DIR)

DOS READY ENHANCEMENTS:

7) BOOT is enhanced
8) CALCULATIONS from Dos Ready in decimal-hex-binary
9) CLEAR is enhanced
1Ø) EDITING of DOS command buffer
11) HIGH MEMORY DEBUG option to allow single-stepping through DOS overlays
12) KEYBOARD DRIVER (full use of Control, Caps & Function keys etc)
13) KEYBOARD Type Ahead option
14) TRSDOS type PARSER
15) PSEUDO SYSTEM FILES (new commands that also operate in Minidos and you can add your own). Including:
16) SETSYS (change system drive from Ø)
17) UNKILL (restore KILLed files)
18) VID (8Øx24 screen option)

Plus assorted utilities such as:

19) DEVICE (allows routing of data to disk file)
2Ø) FILTER (allows filtering of commands)
21) RAM (Ramdisk driver -- Model 4 only at the moment)

PRINTING ENHANCEMENTS:

22) Full MARGIN CONTROL with optional PAGE PAUSE (All models) to format for Basic listings etc
23) PRINTER SPOOLER
24) GRAPHIC SCREEN DUMP TO PRINTER
25) HARD & SOFT FORMFEED CONVERSIONS
26) LINE FEED AFTER CARRIAGE RETURN

CUSTOM BASIC ENHANCEMENTS TO NEWDOS80 DISK BASIC:

PROGRAMMING AIDS:
1) Enhanced Line Editor
2) Enhanced Renumber
3) List Active Variables and their values to screen or printer
4) Enhanced DI - DU
5) Edit direct commands
6) Single Key Keywords Entry
7) Place cursor at error in program line
8) Directly key in characters not on keyboard

ADDED BASIC RUNTIME COMMANDS:

AND$	(logical AND of string)
BEEP	(sound generation)
CALL ADDRESS	(USR function)
CASE...ENDCASE	(added control structure)
CMD"F=POPx	(enhancements)
DO...UNTIL	(added control structure)
#DO	(execute a string as a command)
#DRAW	(draw lines on screen)
ERR$	(print meaningful error message)
#GOTO	(GOTO variable expression)
FCB	(use BASIC FCBs to manipulate disk sectors)
FIELD @	(store string at defined address)
FILL	(fill defined block of memory)
INPUT TO USING	(directly alter strings from keyboard input)
INPUT @	(INPUT @ screen position)
INSTR(!	(bit test -- up to 2040 bits in 255-byte string)
$LET	(aids more efficient use of string space)
LINELET	(allow any chars -- inc quote (") -- stored in string)
LSTRIP	(strip defined characters from left of string)
MOD	(return remainder after integer division)
#MOV	(move block of memory)
OR$	(logical OR of string)
#PLOT	(graphic shape stored in array, turn, magnify)
#RESTORE	(point to DATA line)
#REV	(reverse screen graphics)
ROT$	(rotate characters in string)
RSTRIP	(strip characters from right of string)
SETKEY	(multi-function for single-key entry definition and program definable function key interrupts of program)
SUM	(add up values in array)
#SWP	(swap memory blocks, e.g. screens)
$SWP	(swap characters in string)
UPCASE$	(convert all characters in string to upper case)
VID(exp)	(change screen mode 64x16 - 80x24 - 64x16 etc)
WAIT	(pause for defined key entry)
WPEEK	(word peek or peek two bytes)
WHILE... WEND	(added control structure)
WPOKE	(word poke or poke two bytes)
XOR	(logical XOR of two expressions)
XOR$	(logical XOR of string)

STRING SPACE SAVER background operation minimising garb. collection.

CONTENTS

INDEX

FOREWORD

GETTING STARTED WITH NEWDOS86

CHAPTER 1 ... CHANGES TO DOS COMMANDS AND ADDITIONS

CHAPTER 2 ... AUTOMATIC DISK FORMAT DETERMINATION

CHAPTER 3 ... TECHNICAL DOS INFORMATION

CHAPTER 4 ... PRINTER ENHANCEMENT POINTERS

CHAPTER 5 ... KEYBOARD ENHANCEMENTS

CHAPTER 6 ... INTRODUCTION TO CUSTOM BASIC

CHAPTER 7 ... ENHANCED LINE EDITING (IN BASIC)

CHAPTER 8 ... BASIC UTILITIES

CHAPTER 9 ... NEW BASIC COMMANDS

CHAPTER 10... NEW STRING FUNCTIONS

CHAPTER 11... FORMATTED INPUT ROUTINE (INPUT TO USING)

CHAPTER 12... NEW CONTROL STRUCTURES

CHAPTER 13... NEW GRAPHICS FUNCTIONS

APPENDIX A... SOUND GENERATION

APPENDIX B... EXOTIC DRIVES CONFIGURATIONS

APPENDIX C... NUMBER BASE CONVERSIONS

APPENDIX D... BASIC KEYWORD TOKEN LIST

APPENDIX E... ORIGINAL CUSTOM BASIC COMMANDS (TAPE)

APPENDIX F... USER GROUPS SUPPORTING NEWDOS86

APPENDIX G... HARD DRIVE SUPPLEMENTARY PACKAGE (IF PURCHASED)

APPENDIX H... HELPDISK (SCANNING TEXT FILES FROM ANY MODE)

SUBJECT	CH PG
? COMMAND (CALCULATING FROM DOS READY)	1-19
&%, &!, &H, &D, &O, &B (NUMBER BASE CONVERSIONS)	9- 1
* SUFFIX.	10- 1
ADDED BASIC FUNCTIONS	9- 1
ADDED DOS COMMANDS	1-17
AND$	10- 6
AUTOMATIC DISK FORMAT DETERMINATION	2- 1
BASE CONVERSIONS	9- 1
BEEP	9- 2
BIT TESTING	10- 2
BOOT	1- 1, 1- 5
CALCULATIONS (DOS)	1-19
CALL USING	9- 3
CASE	12- 3
CHANGES TO DOS ROUTINES	1- 5
CHR$64	9- 3
CLEAR *	1- 5
CONFIGURATIONS (EXOTIC)	Ap B- 1
CONTROL KEY DRIVER	5- 1
CONTROL KEY (ELE)	7- 2
CONVERT BASIC PROGRAMS	16- 1, 9-18
COPY	1- 5
CMD"F=POPx	12-10
DATE	1-10
DATE STAMPING (FILES)	1-10
DAY	1-18
DEBUG (HI-MEM)	2- 4
DEC	9- 3
DEF FN	9- 4
DEF USR	9-16
DEVICE	UT14- 1
DI-DU	8- 1
DIR	1- 1, 1-10
#DO	10- 3
DO UNTIL	12- 2
DOS CHANGES	1- 1, 1- 5
DOS KEY FUNCTIONS LIST	5- 3
#DRAW	13- 1
EDIT DIRECT COMMANDS (BASIC)	7- 9
EDIT DIRECT COMMANDS (DOS)	5- 3
ENDKEY	12- 6
ENDCASE	12- 3
ENHANCED LINE EDITOR	7- 1

SUBJECT	CH PG
ENHANCED LINE EDITOR KEY FUNCTIONS LIST	7- 7, 7- 8
ENTERING BASIC (OPTIONS)	6- 1
ERROR (ERR$)	9- 7
EXITING BASIC (SIMPLER)	6- 3
FCB	9- 8
FIELD @	1Ø- 4
FILL	9- 8
FN	9- 4
FORMAT	1- 6, 1- 9, 1-13
FORMAT (OTHER DOS'S)	2- 3
FORMS	1- 1, 1-13
FREE	1-14
FUNCTION KEYS (4/4P)	1- 4
GOSUB	8- 2
#GOTO	9- 9
HARD DRIVES (HARDOS86) (if Supplement Purchased)	Ap G- 1
HELPDISK (SCAN TEXT FILES IN ANY MODE)	Ap H- 1
HIMEM	1- 1
INC	9- 9
INHIBIT ELE	6- 2
INPUT @	9-1Ø
INPUT TO USING	11- 1, UT8- 1
INSTALLATION	FWD- 4
INSTR	1Ø- 5
INSTR(!	1Ø- 2
KEYBOARD DRIVER	5- 1
KEYBOARD TYPE AHEAD (KTA)	5- 2
KEYVAL	12- 6
KTA (BASIC COMMAND)	9-1Ø
$LET	1Ø- 1, 1Ø- 6
LINELET	1Ø- 6
LOGICAL STRING FUNCTIONS	1Ø- 6
LOGON	1-18
LPRINT	6- 2
LSET	1Ø- 8
LSTRIP	1Ø- 7
MASKING PARTSPECS	1-12
MID$	1Ø- 9
MOD	9-11
*MOV	9-12
MULTI-LINE IF.(See CASE)	12- 3
OLDIR	1-18
OR$	1Ø- 6

SUBJECT	CH PG
PARTSPECS	1-11
PATCH FILES	FWD -4
PDRIVE	1-14
#PLOT	3- 2
PRINT @	9-13
PRINTER DEFAULTS	1- 2, 1- 3
PROGRAM SPEED	8- 1, 8- 4,12- 5
PSEUDO SYSTEM FILES	1-17
PURGE	1- 1, 1-16
REDUNDANT AND SHIFTED COMMANDS	6- 3
RENUMBER	8- 3
#RESTORE	9-14
#REV	13- 7
ROT$	10-10
ROUTING PRINTER TO DISK	UT14- 1
RSET	10- 8
RSTRIP	10- 7
SELF-BOOTING ON A 4P (AUTOLOAD MODELA/III ROM IMAGE)	AP B- 2
SETKEY	12- 6
SETSYS	1-18
SHORTHAND BASIC COMMANDS	6- 2
SINGLE KEY ENTRY	5- 5, 7- 3, 12- 6,Ap D- 1
SLEEP	1-18
SORTDIR	1-17, 1-17
SPACES (ELE)	7- 6
SPOOLER	1- 2, 1-13
STRING SPACE SAVER	10-10
SUM	9-15
#SWP	9-16
$SWP	10-11
SYSTEM FILES CHANGES	3- 3
SYSTEM PARAMETERS (ADDED)	1- 2
TECHNICAL INFORMATION	3- 1
TIME	1-10
TRSDOS PARSER	2- 5
UNKILL	1-18
UPCASE$	10-11
USR	9-17
USR ROUTINES (RESERVE SPACE UNDER BASIC)	6- 2
VAR[L]	8- 4
VID (4/4P)	1-18, 9-20, UT13- 1
WAIT	9-21
WHILE WEND	12- 1
WPEEK/WPOKE	9-21
XOR	9- 8
XOR$	10- 6

FUNCTIONS
COMMAND SYNTAX

CONVERSIONS
&H hex$, &D decimal exp, &% decimal exp, &O octal$, &B binary$
LOGICAL STRING FUNCTIONS
AND$(exp$1,exp$2)
TONE GENERATION
BEEP tone, duration: BEEP exp$
CALL M/L FROM BASIC
CALL address USING var%(x)
CONTROL STRUCTURE
CASE:..;condition: code:;condition: code:ENDCASE
SET SCREEN TO 64-CHAR MODE
CHR$64
1Ø USR ROUTINES
var = USR Ø-9(argument)
DECREASE VARIABLE'S VALUE
DEC var%
USER DEFINED FUNCTION
DEF FN var [(parameter list)] = function definition
var = FN var [(argument list)]
DEFUSR Ø-9 = address of routine
TREAT STRING AS DIRECT COMMAND
#DO exp$
CONTROL STRUCTURE
DO:..code..:UNTIL condition
ON-SCREEN GRAPHICS
#DRAWc [x1,y1] TO x2,y2 [TO x3, y3...]
PRINT FULL ERROR IN BASIC
var$ = ERR$(exp)
MANIPULATING FCB'S
var = FCB (exp)
VARIABLE ASSIGNMENT
FIELD @ address, var [LEN exp]
FILL MEMORY
FILL source address,length1,target address,length2
CONTINUE EXECUTION FROM LINE N
#GOTO line

FUNCTION
SYNTAX

INCREASE VARIABLE'S VALUE
INC var%
INSTR FUNCTION
var = INSTR([starting position,] searched$, search$)
BIT TESTING
var = INSTR(!var$,position[,exp])
var = INSTR(!var$1 AND exp$,exp)
INPUT AT DEFINED POSITION
INPUT @ pos,>prompt$;variable list
EXTENDED INPUT
INPUT TO [@pos,] ["prompt string";] var1$ [*] [;] USING var2$
SAVING STRING SPACE
$LET var$ = string exp
LINE INPUT STATEMENT
LINE INPUT [prompt string;] var$
DATA SAVE
LINE INPUT [#-exp,] var$
STORE ANYTHING IN STRING
LINELET var$ = [REM] character string
STRIP CHARACTERS FROM LEFT $
var$ = LSTRIP(exp$[,exp])
CONTROL KTA FROM BASIC
KTA exp$
FIND REMAINDER
var = MOD (exp1,exp2)
MOVE MEMORY BLOCKS
#MOV from, to, bytes to move
LOGICAL STRING FUNCTION
var$ = OR$(exp$1,exp$2)
ON-SCREEN GRAPHICS
#PLOTc var%(Ø) [@(x1,y1)] ,size] ,rotation] [& next parameter set]
ENHANCED PRINT @
PRINT @ [row,column]
RENUMBER PROGRAM
REN [start],inc],first line],last line]

FUNCTIONS
COMMAND SYNTAX

SPECIFIC DATA RESTORATION
#RESTORE line
REVERSE SCREEN
#REV
ROTATE CHARACTERS IN STRING
var$ = ROT$(var$ [,exp])
STRIP CHARACTERS FROM RIGHT $
var$ = RSTRIP(exp$[,exp])
ASSIGN KEY INTERRUPTS
SETKEY exp, var$, GOSUB line
code........ENDKEY
SUMMATE ARRAY
var = SUM var(start)[ntf][STEP value], no. summations
EXCHANGE CHARACTERS IN $
$SWP var$, pos1,pos2,byte length
MOVE MEMORY BLOCKS
#SWP from, to, bytes to move
CONVERT STRING TO UPPERCASE
var$ = UPCASE$(exp$)
LIST OUT VARIABLES
VAR [L] [#] [*] [,list]
PAUSE FOR KEY
WAIT [@exp][>prompt$;][FOR exp$]
CONTROL STRUCTURE
WHILE.condition:..code..:WEND
PEEK TWO BYTES
WPEEK(address)
POKE TWO BYTES
WPOKE address,value
LOGICAL STRING FUNCTION
var$ = XOR$(exp$1,exp$2)
EXCLUSIVE OR 2 EXPRESSIONS
var = XOR(exp1,exp2)

Definition of terms:-

Unless otherwise noted, all parameters may be expressions, which means that they can be variables or literals, or a combination of both. E.g.: 4*A, A(4), CHR$(C)+"4".

var: (variable) refers to a variable only. E.g. A, B, A(4)

Note also that optional parameters are shown in square brackets. [...]

Most CUSTOM BASIC commands requiring numeric parameters will accept values in the range of +/- 93ØØØ before any OVERFLOW errors are reported.

FOREWORD

It is hard to believe that EXTENDED BASIC for '8Ø USERS was released over six years ago. EXTENDED BASIC was originally designed for the tape user owning a 16K machine. The package enhanced Level 2 BASIC almost to the level of a disk based machine, with the exception of the obvious disk I/O routines. EXTENDED BASIC was a BASIC program that compiled a user defined subset of all the available commands and then saved these machine code routines as a /CMD file or SYSTEM tape. To avoid confusion the actual machine code routines were called CUSTOM BASIC for '8Ø USERS.

Even though it started off as a tape based system, it soon became apparent that most of the common DOS's also benefited greatly from the CUSTOM BASIC routines. So I endeavoured to get it to run under most of the popular DOS's. This original package is still available for both tape and DOS users who don't want to use NewDos86.

As time went by, and with a lot of prodding from Kev O'Hare, work was started on making CUSTOM BASIC for '8Ø USERS an intrinsic part of NewDos8Ø BASIC. This allowed me to add further routines and to improve the internal workings of CUSTOM BASIC somewhat.

The original CUSTOM BASIC relied on the production of a SYNTAX ERROR to gain entry into many of its routines. While this mechanism worked reliably, it had a couple of limitations. The major disadvantage, though, was that additional FUNCTIONs couldn't be provided. This new version of CUSTOM BASIC actually tokenises the new keywords. This allows a speedup of execution and also allows the provision of new functions, e.g WPEEK and SUM.

Much work was done to NewDos8Ø, new routines added, routines already present improved. To differentiate between the already excellent DOS, NewDos8Ø, and its superb successor, we decided to give the improved NewDos8Ø the name NewDos86.

How should we handle the manual, now that so many changes had occurred? We originally decided to extend the already existing manual with the new material. This allowed individuals to write software with NewDos86 and then convert it to run on other DOS's or tape by using EXTENDED BASIC to create the required subset of routines.

Now it appears that tape is dead. Most people are using disks, many are using hard disk systems. The manual has been totally reorganised. We have moved the section on the changes to DOS to the front of the manual. Additional material has been inserted where applicable. Information that was only applicable to tape users has been removed. APPENDIX E lists all the routines for tape that have been deleted. This will allow any tape based users out there to see what enhancements are still available.

A major headache was how to treat subjects such as DEFFN and FN. Disk users should be familiar with these techniques. However, feedback from some of the new users associated with the Brisbane TRS-8Ø users group revealed that these chapters were of some benefit. So I decided to leave them in the manual.

New programs are being added to the utilities already written. These will be described in /TXT files on the disk. Check the disk for a README file which will outline any changes and additions that may have been made after the manual has been compiled.

Although many of the problems with memory constraints have been lifted due to overlays being used, the following criteria are still applicable:

Since it is always possible to improve on any routine, the concepts that I kept in mind during the development process were:-

1) To provide machine code routines that you find easy to use and to provide you with enhancements that will ease programming tasks.

2) To keep the size of these routines as small as possible while not compromising the functions they are to carry out, thus freeing the maximum amount of RAM possible for your programs.

Most compromises within the CUSTOM BASIC routines may be overcome by BASIC programming techniques. If you find a limitation that you feel may be better overcome from within CUSTOM BASIC, please let me know and I will investigate the problem further.

Both the DOS and BASIC routines have been extensively tested. However, perfect programs have not yet been written. So, if you have a problem, please let me know, since bug-free code is my aim. It would be more helpful if you could supply the following:-

1) A complete and precise report of the circumstances leading up to the error.

2) Please check that you are in fact using the feature as intended.

3) Endeavour to check that the problem is, in fact, repeatable. If the problem appears to come and go, let me know -- as the testing for such problems will need to be much more extensive.

In fact, I welcome any criticisms and/or suggestions on the routines, or documentation, as it is only by knowing what you, the user, require that I can supply those needs in future enhancements.

A program of this complexity doesn't just happen. Two people deserve a big thankyou. My wife Delma, who has put up with computer talk for many years, and without whose support and encouragement this program would not have come to fruition; and Bill Allen, who has helped greatly with the documentation.

Additionally I would like to thank those people who have helped in providing the many suggestions that have allowed me to fine tune this product. In particular, Jack Decker, Peter Goed and Kevin O'Hare and in general the members of the Brisbane TRS-80 Users group.

Please feel free to contact me at any time. It would be best to contact me the first time via one of the user groups listed in APPENDIX F, as those addresses will remain static.

I would like to make one further point. I suggest that you join a TRS-80 user group if you haven't already done so. All corrections and additional NewDos86 programs will be released via these groups. A list of all groups that I know are supporting NewDos86 are listed in APPENDIX F. This is short at the moment but I hope that it will grow. If your group would be willing to support NewDos86, then contact The Secretary of the Brisbane Group and we will be glad to let you have any information as it comes to hand.

Happy computing,

Warwick Sands.

A couple of points on the documentation:

This printout was made with a DWP410 printer, using TYPE/CMD, a program I wrote and released to the public domain. It is able to apportion spaces equally between words on each line, thus improving on the word processors. The text was prepared using Lazywriter 4+3.

You will notice that some program lines end in a ± while others do not. All the examples included in the documentation were firstly run as BASIC programs and then saved in ASCII and merged into the text file. Some of the early programs were written before the enhanced line editor was incorporated into NewDos86. These end of line markers symbolise the CHR$(0) that BASIC appends to the end of every program line and are explained more fully in Ch 7-2. If you decide to use the standard NewDos80 editor (SYS28/UC) these characters won't appear and if you enter them yourself a SYNTAX error will result.

----------<ND86>----------

** THE NEWDOS86 ENHANCEMENT PACKAGE **

** FOR NEWDOS80 VERSION 2 **

INCORPORATING CUSTOM BASIC ADDONS TO NEWDOS/80 DISK BASIC

----------<ND86>----------

GETTING STARTED

For the Model I/ System 80 (Video Genie) the disk is 40-track, single-density and double-density 40-track for the Models III and 4/4P. Some of the disk/s you have received may be flippies. Please have a look at the file ND86INFO/TXT for any late changes and print all /TXT files to add to the manual.

The disk you have received has a write protect tag on it -- DO NOT remove it. To avoid confusion, this manual will refer to this disk as WSS. The first thing to do is to make a backup copy of WSS, which we will call WSS1. The backup procedure is straightforward if you have two 40 track drives. If your configuration is different, refer to APPENDIX B.

Step 1 Boot with ND86 in drive 0 and type LOGON <ENTER>. LOGON will ask you what sort of drives you have on your system. It is only concerned with the drives, not the diskettes in the drives. So, tell it what you've got (e.g. S40, D40, S80, or D80), and follow the prompts. When it asks you will it update the Pdrives, answer NO. LOGON always updates the Pdrive table in memory and if you answer the prompt, it will also update the Pdrives on the disk. We can't do it in this case as the disk is write protected.

Step 2 Type COPY 0 <ENTER> and follow the prompts. After the copy is completed you will have a backup copy of the disk. This copy we will call WSS1.

Step 3 Put your master disk (WSS) away in safe storage and boot with the new copy (WSS1).

Step 4 is to RUN PATCH/BAS. This program will allow you to customise your copy of NewDos86 to cope with differences between various machines -- e.g. the Model III and the Model 4, which has the extra function keys. To invoke the program enter:-

```
BASIC,+,RUN"PATCH/BAS"
```

You will be presented with a menu, from which you may make several selections, one of which will allow you to install the new keyboard drivers. (There is a lot of drive stepping and noise associated with installing keyboard drivers and it takes a while to do, so don't panic!)

Another choice will allow you to set your SYSTEM options. The program will then re-boot and your ND86 enhancements will be installed (in memory). WSS1 is now your Master ND86 disk.

Step 5 Mount appropriate disks in all drives and execute LOGON. Answer Y to Update Pdrives prompt. Do a PROT Ø RUF and put a write protect tag on WSS1 and treat it hereafter as a Master disk to make working copies only.

Please note that ND86 BASIC (CUSTOM BASIC) is very much improved BEYOND NEWDOS8Ø disk Basic, so the section on the BASIC/CMD changes will require much study. Also note that the disk also contains files SYS28/UC and SYS28/ELE. Since the Enhanced Line Editor requires a machine with lower case and since the ELE removes the NEWDOS8Ø paging feature, I have provided these two files. If you don't have lower case,

COPY SYS28/UC TO SYS28/SYS

This will restore to you the normal NEWDOS8Ø paging functions. (This can also be done if you don't do much BASIC programming and you have no desire to learn how to use the new editor.) If at some time later you fit lower case, simply copy SYS28/ELE to SYS28/SYS and the ELE will then be available.

PATCH/BAS will also patch command files. Various patch files have been provided. The changes are summarised below.

UTILFILE/PDF contains the patches for several NewDos8Ø utilities (list follows). If the file cannot be found a message will be given and the patches for that file skipped:

COMREF/CMD — stops COMREF from writing CHR$(13)'s through the last 16 bytes of the DOS input buffer. These bytes are now used by ND86 to store various parameters and attributes of the DOS. (COMREF is an APPARAT HELP utility. The TRS-8Ø SYSTEM 8Ø COMPUTER USERS GROUP INC.'s HELPDISK system is much better and is provided in this package [See APPENDIX H].)

SUPERZAP/CMD — allows display of the correct Disk Relative Sector value while doing a DFS and when the SPG of the disk <> 5. It inhibits the 8Øx24 driver and allows ASCII display and modification of the data. When in ASCII modify mode, the @ key is used to terminate modification while <shift clear> aborts the changes. See Bits & Bytes #47 P2 -- TRS-8Ø SYSTEM 8Ø COMPUTER USERS GROUP INC., Brisbane publication.

DIRCHECK/CMD — allows handling of disks whose SPG <> 5.

DISASSEM/CMD displays 24 lines when the 80x24 driver is active.

EDTASM/CMD displays 24 lines when the 80x24 driver is active. Additionally, it allows the use of the @ character in labels. Also Binary constants are now valid.

LMOFFSET/CMD No changes. See below.

In addition, a patch is installed to prevent Model 1 versions of the programs from running on the Model 3 DOS and vice versa.

The following /PDF files patch a single file only.

EDAS/PDF Patches EDAS4.1/CMD (supplied by Misosys) to run correctly under the Keyboard Type Ahead routine. Also the 'Q' command under EDAS now allows any legal Mini-Dos command to be executed. E.g.: Q DIR A

FEDII/PDF Patches FEDII/CMD. Please note that the modified FED has not been fully tested. However, the user who requested the patches has not had any trouble to date. Please let me know of any problems that you may find and I will try to provide fixes.

SPRINTER/PDF MODEL 1 only. Patches DOS so that the HOLMES Engineering SPRINTER is correctly initialised at BOOT. The value fed to the speed up mod is dependent upon the setting of SYSTEM option BJ.

Obviously we expect that you would have the NewDos80 manual and be reasonably competent in using NewDos80 as a DOS. NewDos86 in many instances is much easier to use than NewDos80, but we have not tried to duplicate the information in the NewDos80 manual in this package.

----------<ND86>----------

CHAPTER ONE

OVERVIEW OF CHANGES TO THE DOS

A number of enhancements have been added to NEWDOS8Øv2. These include a Keyboard driver including Type-Ahead, a printer spooler, an Automatic Disk Format Determination routine, full margin control and an optional modified version of DEBUG which (when selected) resides in high memory and allows single stepping through the DOS itself.

Please note: Although you can now obtain DIR's and FREE's of TRSDOS 1.3 diskettes, ND86 is still incapable of handling these files in the standard manner, but it will copy the files properly to ND86 formatted diskettes, so copy over your TRSDOS files to ND86 diskettes and use these copies.

The major objectives involved in upgrading NEWDOS8Ø were: Firstly, that ND86 would still function satisfactorily without any of the high memory routines installed. Secondly, any new features added to ND86 should interfere as little as possible with any currently written /JCL files or programs that do DOS calls. This hasn't been totally achieved and the incompatibilities are listed below:

1) DOS COMMAND — The maximum DOS command is now limited to a length of 63 characters instead of 8Ø. This was needed to provide storage for various parameters.

2) FORMS — WIDTH= is no longer a valid parameter. CH 1-13.

3) PURGE — The USR parameter is no longer allowed, but PURGE is greatly enhanced. CH 1-16.

4) DIR — In the STANDARD version, specifying an extension automatically activated both I and S options. In ND86, due to the implementation of partspecs, only the I option is activated. If you wish to see the SYSTEM files, the S option MUST be specified. CH 1-1Ø.

The routines are selected by means of additional SYSTEM options, and are installed at BOOT. SYSTEM option AP (HIMEM) is supported, and the routines are loaded from this point downward in memory at BOOT. If the <shift> key (Clear or F2 is preferable on a 4P) is held down while BOOTing, no routines are installed in high memory and, aside from the changes noted, you are left with a standard NEWDOS8Ø SYSØ/SYS (buffer length is still only 63 characters). If the <D> key is held down at BOOT, the starting address of each routine installed is displayed.

If the <shift> key is not held down, then, after the normal NEWDOS86 initialisation has occurred, a routine is loaded from 52ØØH to 5AØØH which then installs the DOS enhancements in high memory. If the <shift> key is used at BOOT, then the memory from 52ØØH to 5AØØH is NOT used. This allows you to DUMP this memory region to disk if there is a need to preserve it and later re-examine it.

ADDITIONAL SYSTEM PARAMETERS

The following have been added to the standard SYSTEM options or changes made to give even more flexibility for the user to customise his enhancement package to suit the hardware and needs:

BJ = value is changed. Whenever the value is greater than 1, a bit is set indicating that a fast clock is to be used. (In the Model 3 the clock is actually set to the 4Mhz clock. Specifying BJ=1 causes a 2 Mhz clock speed to occur at BOOT.)

BO = y/n BO = Y causes a line feed after carriage return. Very handy for those printers that normally have their line feed turned off on the printer to handle underlining by two-pass in a word processor, thus necessitating software to provide the line feeds.

BP = y/n BP = Y causes all formfeeds to the printer to be converted to the required number of CHR$(13)'s.
BP = N causes all characters to be transmitted unchanged.

BQ = y/n BQ = Y sets margins active.

BR = y/n BR = Y activates a printer driver that allows <break> to abort printing and also allows CHR$(Ø)'s to be sent to the printer. This is forced to Y if BO=Y or BT=Y or BQ=Y or BS<>Ø.
See the Technical Package for details regarding your own routines.

BS = value sets the size of the spooler buffer. BS = Ø disables the spooler. <HJK> purges the buffer, while <FGH> allows/inhibits the sending of characters to the printer.

BT = y/n BT = Y causes the keyboard type ahead feature to be loaded and enabled.
<control 3> toggles the type ahead on and off.
<control 1> reinitialises the KTA and clears the buffer.
<control 4> allows text to be entered into the buffer, but no characters are returned until <control 4> is again hit. BT=y forces CK=y. If your machine has no CTRL key, read CLEAR for control.)

BU = y/n BU = Y enables the display of the type ahead buffer in the top right of screen. <control 2> toggles this feature.

BV = value — value is the default paper length in print lines. The value specified is loaded into 4Ø28H at boot and is used in determining the number of linefeeds to give when a formfeed is requested.

BW = value1 — defines the left margin (value1 = characters to skip)

BX = value2 — defines the right margin (value2 = last character in line to print) -- e.g., 72 gives an 8-char. right margin in an 8Ø-column printer.

BY = value3 — defines top margin (value3 = lines to skip from Top Of Form)

BZ = value4 — defines bottom margin (value4 = last line to print before Form Feed)

CA = value — defines the default maximum print width in characters for your printer. If margins are not active, then the character count is reset to zero when the CA value is reached.

CB = y/n — activates an alive character in the top right hand of screen. <234> toggles the alive character on and off.

CC = value — (Ø-9) causes a user defined routine to be loaded into top of memory. This option allows for one of nine user written routines to be loaded at BOOT. To put your own routines into the system you will need the Technical Package, which contains a program, ATTACH/CMD. This installs your routine into SYSØ/SYS. However, there is, in fact, one user routine already installed for your use, described as follows:

The ROM printer driver assumes that whenever a CHR$(Ø) is sent to the printer that the caller desires the current printer status. Therefore it discards the CHR$(Ø) and returns the status. Many modern printers use CHR$(Ø) in their graphic functions, so all the printer drivers in ND86 treat a CHR$(Ø) as a standard character and send it to the printer in the normal manner. However, some programs (e.g. VISICALC) require the old TANDY standard. The CC=1 routine restores this capability.

CD = value — CD <> Ø causes the Automatic Disk Format Determination routine to load and activate. The value of the parameter is the number of read attempts of the directory before the ADFD tries to reset the PDRIVES. If the ADFD is active, then CE=Y is forced.

CE = y/n — CE = Y causes the routine that handles the SPG parameter to be loaded (Model I only). Model III/4/4P users have this routine permanently installed at the end of SYSØ/SYS.

CF = y/n — CF = Y activates the DOS command line editor. If system option CK = N, then the routine won't load.

CG = y/n — CG = Y activates a save screen routine. Save the screen to a buffer using <control 8>. <control 9> swaps the current screen with the buffer. Note that the high memory Debug uses the same buffer!

CH = Y — causes DEBUG to be loaded into high memory at BOOT. This is an enhanced DEBUG that allows single stepping down to 4ØØØH and the current video display is preserved. More explanation Chapter 2-4.

CJ = y/n — CJ = Y causes a CHR$(26) to be returned when the <shift down arrow> key is pressed. CJ = N causes a null character to be returned, and a CHR$(26) can only be produced by the <SHIFT+DOWN ARROW+Z> sequence.

CK = y/n — CK = Y activates a Keyboard driver which allows special characters to be returned through the use of the <clear> or <control> keys. BASIC's Enhanced Editor requires this driver as does EDAS 4.1. System option CK=Y forces AJ=Y.

CL and CM — are two SYSTEM options in all models for the user's special purposes. Both are two-byte storage areas stored in sector 2 of BOOT/SYS. CL is stored in bytes ØDØH-ØD1H, while CM is stored in bytes ØD2H-ØD3H.

CN CO CP — are three new SYSTEM options for the model 4/4P owners only. They store the values returned by the F1 F2 F3 keys and can be redefined by the user by specifying the desired ASCII values. The values default to:-

CN	F1	6ØH	<SHIFT @> or pause
CO	F2	7FH	+ the BASIC end of line marker
CP	F3	ØC8H	<CNTRL H> used by BASIC and DOS edit functions

CQ and CR — are two new SYSTEM options for the latest release with SYS22GFX/SYS (graphics enhancements for JKL) replacing SYS22/SYS. CQ=Y indicates that screen graphics can be dumped in bit-image EPSON mode. CR=Y indicates an early EPSON mode whereby 2ØH has to be added to the graphic character to print the correct symbol. Obviously only one of these options can be set to Y, as they are mutually exclusive.

----------<ND86>----------

CHANGES TO OTHER DOS ROUTINES

BOOT — has been slightly expanded, in that BOOT * will cause memory to be zeroed before the reboot occurs.

----------<ND86>----------

CLEAR * — zeroes memory between 52ØØH and the current DOS high memory setting and nothing else.

CLEAR,*,value — will cause that value to be used as the fill character. Naturally, this defaults to ØØH.

----------<ND86>----------

COPY & FORMAT — have undergone massive changes. Nevertheless, all of the old Newdos8Ø commands work as expected.

[PLEASE NOTE THAT BECAUSE OF THESE CHANGES TO COPY & FORMAT, THE APPARAT ZAPS ARE NO LONGER IN THE ORIGINAL PLACE ON THE DISK. PLEASE REFER TO THE SYS6/ZAP FILE FOR ALL NEW APPLICABLE ZAP LOCATIONS IF YOU HAVE NEED TO ALTER THEM.]

Additionally, they have been changed to cope with the addition of the N flag and SPG parameter.

Normally the format parameter of COPY defaults to Y. However, if the question FORMAT DISKETTE is asked by COPY then the default is changed to N. This means that FORMAT will attempt to display the name and date of the diskette before formatting occurs.

COPY also recognises that the ADFD exists. If the destination disk is to be formatted, the currently existing PDRIVE entry is maintained. If formatting is not required then the ADFD will be allowed to update the PDRIVE entry.

COPY now allows you to use date stamping in the file selection process. E.g.

```
COPY Ø,1,,CBF,>12/1Ø/87
```

will copy only those files whose date stamp is after December 9, 1987.

COPY also now accepts partspec parameters, also the E (for echo to screen) parameter causes screen display of the file currently being copied during a CBF copy. Example:

```
COPY Ø,1,,CBF,E,UPD,USR,.FF*
```

will copy all updated user files, whose first two characters are FF. As each file is written to the destination diskette, the file name will be displayed on the screen.

Automatic Disk recognition routines can cause problems with COPY and FORMAT. Every time NewDos formats a disk, a PDRIVE table containing the FORMAT information is written to the disk. If DPDN is specified, the data is read from the relevant PDRIVE slot of the SYSTEM disk. If DPDN is not specified, then FORMAT constructs the table by extracting some information from the SYSTEM disk and some information from the PDRIVE table as stored in memory.

This normally causes no problems whatsoever in NewDos8Ø, because the PDRIVE table in memory always corresponds with the PDRIVE tables on the disk. With the ADFD routine, under certain circumstances this may no longer be true. The system disk slot may say that the disk in drive 1 is double sided double density, while the memory PDRIVE says that it is a single sided single density diskette. The information from memory is used to format the diskette, while the information on the disk is used by PDRIVE to tell you what the disk is. Thus a single density diskette might be described by PDRIVE as being a double density diskette. This can be overcome by specifying a DPDN parameter when formatting. Alternatively a PDRIVE,Ø,A can be entered immediately before using FORMAT dn,,,,Y to ensure that the memory PDRIVE table conforms to the system disk's table and to bypass the ADFD (with the ,,,,Y).

Or if you prefer; COPY and FORMAT will now prompt you for the drive details. The following commands activate the prompt mechanism.

FORMAT
COPY [dn=tc][,+,selection parameters]

    where dn is the source drive number and tc is the track count of the source disk. The optional + indicates a file-by-file copy and allows you to enter file selection parameters, e.g. /BAS UPD.

If the source drive number is not stated, the user is prompted for it.

COPY

Input Source drive number ?
Input Destination drive number ?

    Hit the number key corresponding to the required drive.

Format Diskette? <Y>es or <N>o ? Y

    The disk must be formatted if you wish to make a NewDos86 SYSTEM disk.

Is it to be a NewDos86 SYSTEM disk <Y>es or <N>o ? Y

    Assume we wish to create a SYSTEM disk. The N case is outlined below.

Boot Strap Step Rate ?

  Ø = 5mS   1 = 1ØmS   2 = 2ØmS   3 = 3ØmS 3

    Select the number corresponding to the desired speed. This prompt only appears if the disk is to be a SYSTEM disk.

Enter the Disk's Name ?
Enter the Disk's Password ?

If the name - password are not entered then the destination disk receives the name - password of the source disk.

<S>ingle or <D>ouble sided ?

This prompt only appears if you have told the system that your drive is double sided. See LOGON and ADFD. Ch 2-2.

<S>ingle or <D>ouble density ?

Enter the density required.

Track Count ?

If no track count is entered then the track count defaults to 4Ø or 8Ø. The maximum allowable track count for a 4Ø track drive is 49 tracks, while for an 8Ø track drive it is 99 tracks. See LOGON and ADFD.

Is the copy - <E>ntire disk <C>lone <F>ile F

The <E>ntire disk copy is a full sector copy during which the DOS modifies the directory and BOOT/SYS.
The <C>lone copy is also a full disk copy and is used when you want a duplicate of the source disk since DOS does not modify BOOT/SYS and the directory. <C>lone copy should be used when copying NON-NEWDOS SYSTEM disks.
The <F>ile copy, copies files from the source to the destination one file at a time. This can be much faster than the other copies if the source disk is relatively empty. This must be used if the source and destination disks have different characteristics. E.g. when copying from Double Sided 4Ø track drive to a Single Sided 8Ø track drive.
If you attempt a full disk copy and you receive a CONFLICT IN PDRIVE DATA error message then the <F>ile option can be used to accomplish that copy.

Select Files <Y>es or <N>o ?

<N>o: DOS copies every file on the source disk to the destination disk.
<Y>es: DOS will prompt you and make a list of all files to be copied to the destination.

NOTE: The last question only appears if you have requested the <F>ile option.

COPY Ø +,/BAS,UPD,>Ø6/12/87,E

Input Destination drive number ?

The source drive is Ø and only the destination drive is requested.

Format Diskette? <Y>es or <N>o ? Y
Is it to be a NewDos86 SYSTEM disk <Y>es or <N>o ? N
     It is not to be a SYSTEM disk. Therefore, the Boot
     Stepping Rate is not requested.

Enter the Disk's Name ?
Enter the Disk's Password ?
<S>ingle or <D>ouble sided ?
<S>ingle or <D>ouble density ?
Track Count ?
     The prompts and their replies have the same meaning as in
     the previous example.

Is it to be compatible with NewDos8Ø <Y>es or <N>o ?
     This question was not asked in the previous example since
     all NewDos86 SYSTEM disks must be compatible with
     NewDos8Ø.

     <Y>es: The disk is totally compatible with NewDos8Ø.
     <N>o: DOS endeavours to make the disk compatible with
        other DOS's, particularly LDOS and/or TRSDOS6.x.x.
        The directory starting location is set at half the
        track count. The SPG and GPL values are set so that
        one physical track always equals one lump. The PDRIVE
        sector is still written to the disk.

***PLEASE NOTE: if you are creating a 4Ø track Single Density
   TRSDOS for the Model 1, you should choose the <Y>es option
   because TRSDOS 2.3 expects the directory to start on
   track 17.

Select Files <Y>es or <N>o ?
     Since the + parameter was specified, <F>ile COPY is
     assumed. The files copied will be those that meet the
     specifications following the +.

In the example given only files having a /BAS extension with a
date stamp after June 11, 1987 and whose update flag is set
will be copied. The E option causes the filenames to be
displayed on the screen as they are copied.

If you don't format the disk and you are doing a file copy this
additional prompt appears:-

Copy to existing files only <Y>es or <N>o ?
     <N>o: DOS will copy all files from the source disk to
        the destination disk.
     <Y>es: DOS will only copy files that already exist on the
        destination disk. No new files will be created on the
        destination disk.

FORMAT
Input Destination drive number ?
Enter the Disk's Name ?
    The default disk name is DataDisk

Enter the Disk's Password ?
    The default password is PASSWORD

<S>ingle or <D>ouble sided ?
<S>ingle or <D>ouble density ?
Track Count ?
Is it to be compatible with Newdos80 <Y>es or <N>o ?
    The above prompts are the same as in the previous examples.

NOTE: DOS will prompt you to mount the required disks. The destination disk name and date is displayed before any action is taken. Hitting the <up arrow> will abort the process at this juncture.

Those users who have had to do single drive copies may have noticed the requirement to mount the SYSTEM disk one extra time. This is required since DOS has to load SYS8/SYS for date and partspec extraction.

----------<ND86>----------

DATE/TIME When prompted for the date and time at boot, hitting <enter> will allow you to bypass the ND86 validation routine if desired. Since date stamping of files is now implemented, you should leave System option AY=Y and SHOULD always enter the date.

----------<ND86>----------

DIR now gives a sorted directory. TRSDOS 1.3 diskettes can now be DIR'ed, if the Pdrive entry is correct. The files are sorted alphabetically before being printed to screen or printer. A beta tester reported that it takes the program about four minutes (!!) to sort 22Ø entries. A public domain directory sort program, SORTDIR/CMD, is provided on the disk. If this is used, then a definite speed advantage will result once the number of active FDE's exceed 3Ø.

When should you re-sort the directory? Since a power down during the sorting process would render the disk totally useless, my own preference is to only sort the disk directory (with SORTDIR) after I backup a disk.

All the standard NewDos8Ø DIR parameters remain with the following additional parameters and changes:

1) When the P option is selected, the files are now printed 5 to the line versus 4 in the standard DOS, thus making more efficient use of the 8Ø columns paper width of the printer.

2) A new M option is used to force the ADFD to update the PDRIVE for the diskette involved. This is particularly recommended for TRSDOS 1.3 diskettes.

3) A new K option allows the display of only KILLed files. No checks are made as to the integrity of the file, use UNKILL to check integrity.

4) A new X option inhibits the sort process.

5) The original DIR command can be accessed by OLDIR [options]. OLDIR * is equivalent to the old FREE.

6) In the DIR A screen print, the attribute section has been changed to display a (P) -- to indicate a PSF file.

7) A new B option displays date stamping instead of file attributes. E.g.: DIR dn,B will display the dates of the last time the files were closed. To filter according to date: DIR dn,>mm/dd/yy,B displays files whose date stamp is

greater than or equal to the date specified. DIR dn,<mm/dd/yy does the converse. DIR dn,<12/Ø1/86,>12/Ø1/86 will return only files with that date.

8) Partspecs are now allowable. This function is very versatile and is fully detailed below. Note that previously when the (/) was specified INVisible and SYStem files were displayed. Now that partspecs are supported, (/.-) force the display of INV files only. If you wish to see SYS files then the S option must be used.

----------<ND86>----------

PARTSPECS
(Can be used with COPY, DIR & PURGE)

A filespec is the name by which a disk file is referenced. A partspec (PARTial fileSPEC) is a name by which GROUPS of files may be referenced. NEWDOS8Ø had a limited form of partspecs in that some commands allowed you to reference groups of files according to the extension (/ext), e.g. DIR /BAS.

This concept has been greatly expanded in this enhanced ND86, and, although simple, it may take you a little while to learn to use partspecs to maximum advantage.

The five symbols which indicate partspecs are:

/ . - > <

(/) The slash is something we're all familiar with. There is no real change here. The slash simply indicates that the following characters specify an extension which must match that of the file. If the extensions don't match, don't print the file.

(.) Period indicates that following is a filename that is to be used. The period is to the filename what the slash is to the extension.

(-) The minus indicates a NOT function. E.g. DIR -/BAS will only list those files that don't have a /BAS extension. The minus also replaces the period.

(>)(<) The greater-than less-than signs allow you to specify ranges. E.g.: >L <O will limit files to those whose first character falls between L and O inclusive (i.e. L, M, N and O).

Let's look at a few real examples. I'll give the complete command for clarity.

DIR >M,/BAS will display only those files greater than or equal to M and whose extensions are /BAS.

DIR >M,-/BAS will display files greater than or equal to M whose extensions are not /BAS.

DIR -BASIC will only list those files whose filename is not BASIC. E.g. BASIC/DAT,BASIC/CMD and BASIC/FF would not be listed.

DIR .BASIC will only list BASIC/DAT,BASIC/CMD and BASIC/FF (in other words, any files named BASIC with whatever extension).

----------<ND86>----------

MASKING PARTSPECS

Partspecs are very versatile, as it is also possible to "mask out" characters within the partspec. A question mark (?) indicates that any character, including no character at all, in that position is acceptable while an (@) indicates that the character must be alphanumeric (i.e. filters out spaces etc). (Actually any name short of 8 characters is padded by spaces in the directory, also the extension when less than 3 chars.) The asterisk (*) indicates that all characters from this point on are acceptable. Examples:

DIR 1 /@@ will list out all files having two-character extensions.

DIR 1 /?? will list out only files having none, one or two characters for the extension.

DIR 1 .?A* will list out all filenames whose second character is 'A'.

DIR 1 -FF*/BAS will list out all files excepting any files having a /BAS extension and whose first two characters are FF.

DIR 1,.BA* will list all files having BA as the first two characters of the name.

DIR 1,.??? will list all filespecs with names less than four characters long.

DIR 1 /B?S M will list all files with 3 character extensions beginning with B and ending with S and will force a PDRIVE re-initialisation if the ADFD is installed.

DIR 1,>E,K will list only those killed filespecs whose first character is greater than or equal to E.

Please try out other examples yourself and become completely familiar with these functions. You will find that it does make life easier -- particularly if you have large directories. Partspecs can be used with COPY, DIR and PURGE.

----------<ND86>----------

FORMAT see changes under COPY heading. Note that the ADFD will endeavour to change the PDRIVE setting so the disk name and date can be displayed. If the disk is unformatted this process can take several minutes. If you wish to bypass this procedure use the Y option, e.g.

FORMAT dn,,,,Y

Note that there must be four (4) commas between the drive number and the Y to define the mandatory parameters -- even if they are nulls. The Y option is also available in COPY.

----------<ND86>----------

FORMS in NEWDOS8Ø, was available only on Model III upwards. In ND86, Model I/ System 8Ø (Video Genie) users also have the facility. The following is now the correct syntax. This applies for all models:

```
FORMS [LEFT=x] [,RIGHT=x] [,TOP=x] [,BOTTOM=x]
        [,COLS=x] [,LINES=x],[,SPOOLER=yz] [,T][,ON][,OFF]

FORMS P[,"literal string"] [,x] [,x]
```

where x is any value between Ø and 255 (ØØH to FFH), while yz must be in the range Ø-3ØØØØ. A value of Ø for left, right, top or bottom turns off the margin concerned. COLS= defines maximum print width in characters (System parameter CA is the default). The margin values are not checked for conflict. It is up to the USER to MAKE CERTAIN that the values are correct.

The SPOOLER= value is checked to see that memory is available. It isn't possible to set a spooler size that would reduce TOPMEM below 35ØØØ. The SPOOLER parameter may be changed from BASIC provided that BASIC has reserved no high memory . Since it isn't feasible for FORMS to change the position of BASIC's STACK, it should be immediately followed by a CLEAR 5Ø statement (The CLEAR statement must have an argument). This will move BASIC's stack out of the buffer area.

FORMS ON Enables the Margin routine
FORMS OFF Disables the Margin routine

This allows label programs to temporarily disable the margins without having to set a great number of parameters.

The 'T' option causes a CHR$(12) to be sent to the printer. This normally forces the printer to give a page feed.

The P option must be the ONLY parameter when specified. It allows data to be sent to the printer. Decimal or hexadecimal values are allowed and the literal string must be within quotes. Note that ND86 forces all text to uppercase. Therefore:

```
FORMS P,"A simple test",10,13
```

will cause "A SIMPLE TEST" to be sent to the printer, followed by a linefeed and a carriage return.

-----------<ND86>----------

FREE — will work with any TRSDOS 1.3 diskettes mounted in drives 1-3.

FREE :dn — will give a display of the used grans for that drive. A free gran is represented by a period, a used gran by an 'x' while the directory grans are represented by 'D'.

-----------<ND86>----------

PDRIVE — has been changed. Two extra parameters are now available. The first is the option to specify Sectors Per Gran (SPG=). This allows ND86 to format DOUBLE-DENSITY disks that are readable by other DOS's. E.g.

```
PDRIVE,0,1,TI=x,TD=E,TC=40,SPT=18,SPG=6,GPL=3,DDSL=20,DDGA=3
```

will allow NewDos to format a double-density single-sided disk compatible with LDOS, Multidos etc. Note that it may be necessary for the other DOS to do a REPAIR on the disk to correct their 'PDRIVE' tables stored in the directory. For single density, SPG=5 and GPL=2 is the standard. If SPG=0 then SPG=5 is assumed.

Users who are working with a great number of small files may like to try formatting their data disks with SPG=3,GPL=6,DDGA=6. This helps to minimise the wastage of unused sectors in the last gran of a file. With SPG=3, DDGA must be at least 4 to ensure the directory size is at least 10 sectors. Please note that if an SPG value other than 5 is used, NewDos80 will NOT be able to read the disk.

The second parameter is TI=N. This indicates that the disk is not a ND86 disk. This parameter is used to determine how many Directory sectors to read. In NewDos, the sector count is computed from byte 1FH in the HIT sector, whilst other DOS's assume that the directory occupies a full track. This parameter is set by the ADFD routine if the BOOT sector isn't ND86. Flag M implies flag N.

For the technically minded: The N flag is stored in Bit 5 of Byte ØEH of the PDRIVE slot and is reflected in bit 5 of Byte 7. The SPG value is stored numerically in byte ØBH of the slot. However, since there isn't a spare byte in the PDRIVE slot in memory, this byte is combined with byte 5 (GPL). To maintain compatibility with NEWDOS8Ø: 5 is first subtracted from the SPG value, and this is then stored in the top nybble of byte 5. Assume GPL=2, then if SPG=5, byte 5 = Ø2; if SPG=6, byte 5 = 12. A small routine intercepts the PDRIVE initialisation, gets the SPG value and stores it within DOS and also resets the GPL to its correct value.

Just an extra point on PDRIVES. The other DOS's do have a PDRIVE table equivalent -- they just won't let you change any of the parameters. LDOS stores its PDRIVE information in the GAT sector of the directory, which is why a REPAIR ALIEN must be done when you format with ND86.

----------<ND86>----------

PRINT,filespec,* will print out the entire file without doing any character value manipulation. SYSTEM option AX=xx has no effect.

This allows you to route a file destined for the printer to a disk file, and send the file to the printer at a later date.

Mention should be made at this point of the utility TYPE/CMD. This is a full featured formatted text print routine. See TYPE/TXT for more details.

----------<ND86>----------

PURGE has undergone major changes. PURGE,dn,USR is no longer valid. Rather the options following PURGE are now similar to those following DIR.

PURGE,dn,I,S,U,partspec,datestamp

The responses in PURGE also have been extended to include <A> for all valid files. Hitting <A> will cause all files from that point onwards to be killed. Note that the <N> and <Q> responses still work, but as there is no delay between the questioning prompt and the killing of the file, you will have to be quick, since Y is otherwise assumed.

Examples:

PURGE,1

will query for only the normally visible files on the disk.

PURGE,1,U,>B,<M,>1Ø/12/86,<12/11/86

will query for updated files between B and M inclusive, and dated between Oct 12, 86 and Dec 11, 86 inclusive.

PURGE :1 -/BAS

will query for all visible and invisible files that don't have the /BAS extension.

PURGE,1,I,S

performs identically to the old NewDos8Ø PURGE command. It allows every file on the Disk to be killed, with the exception of BOOT/SYS and DIR/SYS.

----------<ND86>----------

ADDITIONAL DOS COMMANDS

A feature new to NEWDOS8Ø, and most other DOS's, is the ability for you to be able to incorporate your own PSEUDO SYSTEM FILES (PSF) into the DOS. These files need not reside on the SYSTEM disk and can be executed from Mini-Dos. While this feature is very powerful, it is also very dangerous if not set up as follows:

These files must reside in an area of memory that won't affect the main program. An obvious area is the DOS overlay area (4DØØH-51FFH). After you have assembled the file, (or if you copy it from another disk), you must give it the PSF attribute. This is done with the ATTRIB command:

ATTRIB filespec,PSF=Y

This must be done!! If you attempt to execute a file that loads in the DOS overlay region and it doesn't have PSF status, the system may lock-up or worse. To see these PSF files, the S option must be specified in DIR.

A disadvantage in using the DOS overlay area is that the DOS file handling routines also use this area. Therefore file handling is slightly more involved. Full details are available in the Technical Package. These difficulties can be removed by reserving your own overlay area in high memory by setting SYSTEM option AP, or if the file is to operate under BASIC, the overlay areas 52ØØH-56E7H and 6FØØH-73E7H are also available.

A small comment regarding the way NEWDOS copies files: When a full disk copy is done, the destination file takes on the attributes of the source file. However, when a single file copy occurs, the destination file retains its attributes. This has particular application to PSF files.

If you do a single file copy of a PSF file the destination file will have to be given the PSF attribute. If in doubt do a DIR A on the file concerned and check that it does have the PSF attribute.

I have included quite a few public domain programs that have been adapted for PSF working. Rather than listing them all here, see the separate sections on UTILITY files for full details. Also there may be some new files documented in the README/TXT file -- don't forget to print them out also, as they contain important information. The current list of PSF's follows on next page:

DAY prints the full date on the screen. Also see CAPTURE, UT1-1. Capture is used to store the expanded DAY format into a Basic program to print into documents.

LOGON [:dn]allows you to tell the SYSTEM of changes in disk drive specifications. It then performs an equivalent of FREE, but forces an update of PDRIVEs for all mounted drives. If the optional drive number is specified, then only the PDRIVE entry for that drive is updated.

OLDIR This is the original ND8Ø DIR & FREE. Since the DIR of NewDos86 sorts the directory, it has to read the entire Directory before any printout is made. If there is an unreadable directory sector, a DIRECTORY READ ERROR will be given before any files are displayed. OLDIR, on the other hand, prints out the directory a sector at a time. If after using LOGON you still cannot DIR a diskette use OLDIR. OLDIR * does the equivalent of FREE.

SETSYS/CMD Allows you to specify which disk is the SYSTEM disk. Some DOS's force the SYSTEM disk to be mounted on drive Ø. ND86 does not do this, it simply tells the DOS that drive x is now the SYSTEM drive. The drive numbering does not change. SETSYS checks that SYS1/SYS is available on the new SYSTEM drive. If SYS1/SYS cannot be found, you are prompted to hit <enter> when a SYSTEM disk is mounted. Hitting a number key at this point will select that drive as the SYSTEM drive.

SLEEP clears the screen and suspends all DOS operations until any key is hit on the keyboard. The screen contents are preserved. SLEEP,Y doesn't prompt, but executes immediately.

SYSRES/CMD allows a variation on the way a RamDisk can operate. It allows selected SYSTEM files to be stored on and retrieved from the RamDisk. See UTILITY-5 for more details.

UNKILL,[$] filespec will restore a KILLed file, if possible. The optional $ in the UNKILL command is to allow prompting for the mount of a target diskette. UNKILL will abort if the file has been corrupted.

VID initialises 8Øx24 screen display mode. (For Models 4/4P only)

Hopefully, as programmers become aware of this PSF feature, other public domain programs will be adapted to reside in the DOS overlay area.

----------<ND86>----------

CALCULATIONS FROM DOS READY

THE ? COMMAND

The ? command was designed for lazy programmers and allows you to do simple arithmetic functions whilst at DOS or Mini-Dos ready. The routine was not designed to replace your 16 digit HP! There are 3 variations on the use of the command.

?,exp will display the result of exp in Hex, Decimal and Binary formats.

?,P exp will display the contents of the address specified by exp. It is essentially a PEEK function.

?,PW exp will display the contents of the two bytes starting at the address specified by exp. It is a WORD PEEK.

The valid operators within the expression are:

```
+     addition
-     subtraction
*     multiplication
/     division
!     OR
&     AND
#     XOR
%     MODULUS
```

There is no hierarchical precedence, the expression is evaluated strictly from left to right. No errors are given if a mathematical error has occurred. If a value greater than FFFFH is input or you try to divide by zero, a BAD PARAMETER error will result.

You can enter values in Hex, Decimal or Binary. Since the letter B is a valid hex number, and since a binary number is indicated if the first character of the expression is B then if you are inputting a HEX number beginning with B then you will need to have a leading Ø. Examples:

```
?, b1Ø1 + b1Ø1Ø  ->  ØØØFH = 15 =  1111 1111 :
?, B1Ø1 + 1Ø     ->  ØØØFH = 15 =  1111 1111 :
?, Øbh + 2       ->  ØØØDH = 13 =  ØØØØ 11Ø1 :
?, f3h           ->  ØØF3H = 243 =  1111 ØØ11 :
```

The binary display drops the leading word if it is ØØH. To avoid confusion when displaying 4 hex bytes, the two groups of 16 binary digits are separated by a colon.

----------<ND86>----------

CHAPTER TWO

AUTOMATIC DISK FORMAT DETERMINATION ROUTINE -- CD <> Ø

This routine was written so that you, the user, will have to worry about the PDRIVE table as little as possible. This routine requires about 5ØØ bytes and lives in high memory. Whenever an error results while the disk directory is being accessed, this program will endeavour to reset the PDRIVE table of the drive concerned.

Note that this routine allows you to swap system disks to your heart's content. Single or double sided, single or double density, providing your drive can read it, you can swap it.

There are some situations that it cannot cope with. This is where disks have been swapped in a drive and the new disk has the directory at the same physical location as the old disk, but the new disk has a different GPL value or has a different number of sides. This won't happen often, but it can occur. Also Model III and 4/4P users please note that sometimes a disk may have to be WRDIRPed before it can be read.

If you wish to ensure that the PDRIVEs are correct, do a DIR,dn,M or a LOGON. The routine has been designed to cope with the 'TRSDOS' standard, i.e. the third byte of the BOOT sector contains the Directory starting track. There appear to be DOS's coming on to the market not conforming to this standard. These are not supported.

The best suggestion I can make if ADFD fails to read a directory, is to use SUPERZAP's VDS function to determine where the diskette's directory is, by making SUPERZAP pause when a read protected sector is encountered and then computing the DDSL, by dividing the relative number of the first protected sector by 18 [or 36 if a double sider], then 18 [or 36] divided by the SPG to determine the GPL value to try -- problems of this nature should only occur on double density disks, if at all.

The DOS uses the GPLxDDSLxSPG firstly to figure which sector starts the directory, so you can manually help the DOS by changing parameters in the PDRIVE specs which will allow it to compute the correct sector number for a "strange" disk.

Since the CD value replaces the normal AM retry parameter while the directory is being read, if you experience a spate of DIRECTORY READ ERRORs, try increasing the CD value. If this 'fixes' the problem, then it is probably time to backup that disk and then throw it away -- as it no longer can be trusted as a reliable storage medium.

The routine itself is accessed whenever a DIRectory sector is to be read. If the sector can be read without error the routine exits. This can be a problem when trying to access TRSDOS 1.3 diskettes since the only sectors that aren't identified as 'directory' sectors are those of the directory itself, which is why the 'M' option was included in DIR.

If an error occurs, the routine loads the BOOT sector. During the load of the boot sector, the routine has determined the disk's density, whether double stepping is required, and whether the track/sector numbering starts at Ø or 1.

If the disk has been formatted by NewDos, the PDRIVE sector (sector 2) is loaded and the first 1Ø bytes of that sector are copied down into the PDRIVE entry concerned. (When NewDos formats a disk, it places the appropriate PDRIVE information in the first 16 bytes of that sector.) If you can't read a NewDos formatted disk, then possibly the boot sector has been corrupted or the wrong PDRIVE information has been stored on the first 16 bytes of relative sector 2 of that disk.

Otherwise, if the drive is double-sided, the disk is checked to see if it is a double-sided diskette. If the drive is an 8Ø-tracker, the routine tries to read the disk as an 8Ø-track disk. If this fails, the disk is assumed to be a 4Ø-tracker.

Obviously, all this disk checking takes time. The worst case is a single density, single sided 4Ø-track disk in an 8Ø-track double-sided drive with a double-density system. This takes about 35 seconds.

There are several points to note about this routine. At boot-up, the initialisation process checks the PDRIVE tables of the active drives and uses this information to adjust the recognition routine accordingly. The initialisation checks for the following information:

1) If the first four drives are specified as single density, then the initialisation routine assumes that your system doesn't support Double Density.

2) The routine then determines which drives are Double Sided (TD = C or G);

3) and which drives have 8Ø-track capability (TC > 63 or TI=L).

This was done to prevent the routine from trying to accomplish the impossible -- such as trying to read Double Density in a Single Density system or trying to double step a 4Ø-track drive or read the non-existent back side of a single-sided drive.

What it does mean is that YOU, the user, must have the PDRIVEs of your booting disk set up correctly in the first place. If, for some reason, you have to boot with a single-sided disk on a double-sided drive, LOGON allows you to tell the system your true hardware configuration. This remains valid until the next reboot.

If bit 5 of 4359H (4266H MOD III) is set, the routine resets the bit and forces a PDRIVE update. The 'M' option of DIR sets this bit. If bit 7 of 436BH (428BH) is set, the ADFD is disabled. DOS clears this bit when returning to DOS (or Mini-DOS) ready, COPY and FORMAT use this bit to ensure that the PDRIVE tables are not corrupted during a format. The routine doesn't alter TSR values or type of interface information stored in TI (flags A to E).

8-inch disks not formatted by NEWDOS are not catered for at present.

If the disk isn't formatted by NEWDOS, then the following assumptions are made:

For the standard TRSDOS 2.3/LDOS/MULTIDOS format:

1) For Double Density, GPL = 3 and SPG = 6. For Single Density, GPL = 2 and SPG = 5. If the drives are double sided, then the GPL is doubled.

2) One track = one lump. And therefore the directory will always occupy one track on the disk.

3) The directory track is stored in the 3rd byte of BOOT/SYS. VTOS is an exception, where it is stored in the 4th byte.

For TRSDOS 1.2/3

1) For Double Density, GPL = 6 and SPG = 3. For Single Density, GPL = 2 and SPG = 5. Double sided drives are not supported by TRSDOS 1.2/1.3

2) One track = one lump. And therefore the directory will always occupy one track on the disk.

3) The directory starting track is always at track 17.

----------<ND86>----------

IN-MEMORY DEBUG -- CH=Y

This is the standard NewDos8Ø DEBUG with a few small modifications. The video screen is saved when you enter DEBUG and restored on exit and the KTA is disabled. Holding down the <enter> key will display the saved screen. DEBUG won't exit until the <enter> key is released, allowing routines which get a character from the keyboard to function correctly. DEBUG always uses the ROM routine to get a character, allowing the successful modification of Keyboard routines and vectors. This means that auto key repeat is not available.

The TRS-8Ø SYSTEM 8Ø Computer Group in Brisbane has designed a very comprehensive 'HELP' file and handler to run under NewDos8Ø/86 called HELPDISK and is incorporated with this package (See Appendix H). In ND86, if the <shift key> is held down with <123>, DEBUG is entered, otherwise entry to the 'HELP' file handler occurs (CH=N).

Since the in-memory DEBUG occupies around 25ØØ bytes, the only time it would be activated is when serious debugging is desired. However, entry to the HELPDISK operation is still possible with the in-memory DEBUG and is achieved if the <shift key> is depressed along with the <123> keys (CH=Y).

----------<ND86>----------

THE TRSDOS PARSER

Most TRSDOS compatible DOS's have provided a routine which would PARSE an optional parameter string. This routine is called by most /CMD programs written for the TRSDOS environment. In TRSDOS 2.3 this routine resided within SYS1/SYS, and NEWDOS replaced it with the Mini-Dos function. I have now incorporated this routine in SYS24/SYS.

So if your documentation said that parameters were allowed to follow the command line, but every time you tried you received an ILLEGAL DOS FUNCTION error, now you know why. Try it again with this enhanced ND86.

For all you machine code programmers out there, if you wish to use it to parse your command line, feel free. The line to parse may be enclosed in parentheses, but these are not necessary. The parameter may be specified by up to a six-character name. Acceptable formats are:

param=X'nnnn'	Hexadecimal
param=nnnnH	Hexadecimal
param=dddd	Decimal value
param="string"	alphanumeric data
param=flag	ON,OFF,Y,N,YES,NO

The addresses to call the routine are at 4476H in the Model I and 4454H in the Model III. These are TRSDOS compatible addresses. Let's now examine how the routine is called.

On entry:

HL -> the command line to parse
DE -> the beginning of your parameter table

On exit:
The flags:
Z is set if no parameters found or only valid parameters
NZ if a bad parameter was encountered

Your parameter table must have the following format:

The name field is six characters long. If the parameter name is less than six characters, then spaces must be used to fill the field. This is followed by a two-byte address of the WORD location where your parameter value will be stored.

This format can be repeated indefinitely and the end of the table is indicated by a ØØH byte.

Numeric values are evaluated and the result stored in the memory address specified.

String inputs. The address of the character following the double quote (") is stored in the specified location, the closing double quote is replaced with a Ø3H byte, this makes it acceptable to ND86 routines which open and close files etc.

Flag values. The entries Y, YES and ON return a value of ØFFFFH. While N,NO and OFF return a value of ØØØØH. If a parameter is specified without a following (=), then Y is assumed, whereas if the (=) is included then N is assumed.

Example:

```
USER TABLE
       DEFM    'HIMEM '            ;first parameter
       DEFW    8ØØØH               ;store value at 8ØØØH
       DEFM    'File  '            ;second parameter
       DEFW    8ØØ2H               ;address 2
       DEFM    '/ext  '            ;default extension
       DEFW    8ØØ4H               ;address 3
       DEFB    Ø                   ;terminate the table
```

-----------<ND86>----------

CHAPTER THREE

TECHNICAL INFORMATION

The following bytes in the CB Communications Region (CBCR) have been set aside for the following purposes.

Model I	Model III	FUNCTION	
4358H	4265H	Bit map	
	BIT	Installed routine	SYSTEM option
	Ø	SPG routine	CD <> Ø
	1	Alive character	CB = Y
	2	High memory DEBUG	CH = Y
	3	Printer margins	BQ = Y
	4	Alternate printer driver	BR = Y
	5	Printer Spooler	BS <> Ø
	6	User Printer Driver	CC <> Ø
	7	CKD routine	CK = Y
4359H	4266H	Bit map	
	Ø-4	Reserved future use	
	5	force update of PDRIVE table	
	6	screen save routine	CG = Y
	7	keyboard type ahead	BT = Y
435AH	4267H	Left margin	
435BH	4268H	Top margin	
435CH	4269H	bottom margin	
435DH	426AH	Size of spooler buffer	
435FH	426CH	Address of alternate keyboard table	
4361H	426EH	Pointer to start of Pdrive table	
4363H	427ØH	If 1, then 8Øx24 video mode enabled	
4364H	4271H	8Ø-track drives in 1,2,4,8 format	
4365H	4272H	Double-sided drives in 1,2,4,8 format	
4366H	4273H	Status byte for Spooler control	
	BIT	Function	
	Ø	Character is being stored	
	1	Character is being sent to printer	
	2	Buffer has contents	
	3	Not used	
	4	Spooling paused	
	5	Not used	
	6	Disable spooler - user	
	7	Disable spooler - control	
4367H	4274H	Status byte for Keyboard Type Ahead	
	BIT	Function	
	Ø	Keyboard scan active	
	1	Interrupt active	
	2	Buffer full	
	3	Buffer has contents	
	4	Spooling on	

5	Display type ahead
6	Disable KTA - user <^3>
7	Disable KTA until next keyboard scan.

The status bytes for the KTA and spooler have been designed so that storing a zero byte therein will 're-initialise' the spooling routines.

Just for completeness:

4Ø28H	Maximum lines per page + 1
4Ø29H	Number of lines printed on page + 1
4Ø2AH	Number of characters printed on line
4Ø2BH	Maximum number of characters per line (right margin)
4Ø2CH	Maximum paper width

4369H	4289H	Bit Ø - BASIC active
		Bit 1 - use alternate KB table

4423H Keyboard Control chr scan.

Every time an interrupt occurs, the control keys are scanned and the following information is stored into 4423H. Note that no clearing of 4423H is done by this routine. It is therefore the user's responsibility to zero this byte.

Bit	
7	Indicates that the byte is being updated
6	<shift right arrow>
5	<shift left arrow>
4	<shift down arrow>
3	<shift up arrow>
2	<enter>
1	<shift @>
Ø	<break>

This byte is the same in both models I and III. BASIC uses this byte to check for <break> and <shift @> keys; as does EDAS 4.1

Bit 5 of byte Ø of a file's FPDE is set if that file has the PSF attribute.

If you always use the pointers in RAM to determine start of BASIC programs etc, you will have no trouble adapting your programs. Please add one further pointer to your list -- 66BEH will now point to the start of BASIC's file buffers. Previously, Basic file buffers used to start at 66BEH. These have been moved to 74ØØH to make space for the Custom Basic enhancements.

Pointer to Basic buffer for keyboard entry: 4ØA7H
Pointer to start of BASIC's file buffers: 66BEH

----------<ND86>----------

CHANGES TO SYSTEM FILES

Laid out below are the changes that have been made to the /SYS files.

SYSØ/SYS	small changes
SYS1/SYS	small changes
SYS2/SYS	small changes
SYS3/SYS	major changes to PURGE, now contains APPEND
SYS4/SYS	small changes
SYS5/SYS	no changes
SYS6/SYS	major changes for Auto recognition, SPG and partspecs
SYS7/SYS	small changes
SYS8/SYS	major changes to DIR
SYS9/SYS	small changes
SYS1Ø/SYS	handles GET and PUT, small changes
SYS11/SYS	handles RENUM, small changes
SYS12/SYS	handles REF, major changes
SYS13/SYS	handles ERRORS and Edit, major changes
SYS14/SYS	small changes
SYS15/SYS	FORMS,SETCOM,DATE,TIME
SYS16/SYS	minor changes
SYS17/SYS	SYSTEM, minor changes
SYS18/SYS	BASIC Enhanced Line Editor
SYS19/SYS	LOAD, SAVE, RUN etc., minor changes
SYS2Ø/SYS	BASIC functions, major changes
SYS21/SYS	CMD"O", no changes
SYS22/SYS	JKL, Dos Command Editor
SYS23/SYS	REN, DI, DU, RENUM
SYS24/SYS	Parser, ?, LIST, PRINT
SYS25/SYS	Graphic commands, SETKEY
SYS26/SYS	INPUT TO, FIELD@ etc
SYS27/SYS	BASIC CMD"functions
SYS28/SYS	BASIC direct command mode handler
SYS29/SYS	NCB subroutine file

Note that the /SYS files from 21 to 25 load in the 4DØØ-51FFH DOS overlay area. /SYS files 26 and 27 use the NCB buffer 1, SYS29 uses NCB buffer 2, while SYS28/SYS uses NCB buffers 1 and 2. The 24 bytes following the end of the overlay areas are used for storage of various parameters.

CUSTOM BASIC overlay area #1:	6AØØ - 6EE7H
CUSTOM BASIC overlay area #2:	6FØØ - 73E7H

Notes on reduced systems.

1) COPY dn1,dn2,,CBF... now uses SYS8/SYS and SYS17/SYS.
2) DIR and FREE uses SYS8/SYS and SYS24/SYS.
3) PURGE uses SYS8/SYS and SYS3/SYS.

----------<ND86>----------

CHAPTER FOUR

PRINTER FUNCTIONS

There are enhancements to the Newdos8Ø Model III version Dos command FORMS to give you greater control of the printer. -- See CH 1-2 for default values from System options, also CH 1-13 for full description of the new FORMS. FORMS is also available in the Model I version of Newdos86.

An in-memory spooler is available. -- See added System Option BS, CH 1-2.

One of the new PSF files is PAGE/CMD, which allows you to cause printing to pause at each form feed if desired. -- See Utility 4.

DEVICE/CMD allows you to route data from the printer to a disk file. -- See Utility 14.

Incompatibilities between machines in port addressing for printing purposes can occur with some applications software. ND86 will install the correct printer driver for its own purposes, but for the users' information the following are the printer ports used by NEWDOS86 for the various model machines:

```
Model I                   -- 37E8H
System 8Ø, Video Genie -- ØFDH
Model III,4,4P            -- ØF8H
```

For completeness, what follows are the actual hardware addresses used by the various machines:

```
Model I                   -- 37E8H

System 8Ø, Video Genie -- ØFDH (4Ø1Ø Expansion unit),
                          ØFDH and 37E8H (4Ø2Ø Expansion unit).

Model III,4,4P            -- ØF8H and 37E8H
```

----------<ND86>----------

KEYBOARD FUNCTIONS

THE <CONTROL> KEY DRIVER -- CK=Y

The CKD is required by both the Enhanced Line Editor in BASIC, the DOS editor and EDAS 4.1. It allows normally unobtainable characters to be returned from the keyboard through the use of a <control> key. The <control> key will be signified in future by <^>, (Model I, Model III and SYSTEM 80 (Video Genie) use the <CLEAR> key as the control key. Model 4/4P owners use either <CTRL> or <CLEAR> key.) Thus <^ H> is obtained by holding the <control> key down and then the <H>. The <shift> key will be represented by <|>. Thus <| ^> is the <shift> key pressed then the <control> key pressed, while <^ |> is first the <control> key and then the <shift> key.

Note that <clear> no longer returns a CHR$(31) -- <| clear> does instead in order to release <clear> for <control> use. Combinations of <^> and <^|> together with ,./; and <enter> will return the normally inaccessible characters. Experiment and ye shall know. For other keys the ASCII value returned is that of the key pressed plus 128, i.e. Bit 7 is set. Note that <^1> to <^0> are reserved for control purposes, so avoid using these in your programs.

Model I, III and SYSTEM 80 users, please remember to read <clear> when you see <control> or <^>.

Some programs, such as EDAS 4.1, expect a different character to be returned when the arrow and control keys are pressed. The LDOS manual calls this Extended Cursor Addressing (ECA). Whereas most of the characters returned from the keyboard are derived 'mathematically' from their position in the matrix, the arrow keys and control keys (<enter>, <break> etc) are read from a lookup table. (This resides in ROM at 50H.) To achieve ECA, a substitute table is used.

My CKD allows for an alternate table. The address of this table must be stored at 435FH (426CH Mod III), and is used when bit 1 of 4369H (4289H Mod III) is set. The Enhanced Line Editor in BASIC uses this technique. This bit is reset when return is made to DOS ready.

CKD INCOMPATIBILITIES

To overcome incompatibilities between the Control Key Driver and other programs, such as ALLWRITE and MODEM 8Ø, a small routine has been added that allows depressing <9Ø:> to disable/enable the CKD. <9Ø:> enables the CKD, while <shift 9Ø:> disables the CKD.

----------<ND86>----------

TOGGLING FUNCTIONS ON/OFF

This convention applies to all routines where a group of keys perform a toggling function. The keys alone will enable the function. If the <shift> key/s are held down as well, the function is disabled.

This affects:-

<234> - the Alive Character)
<9Ø:> - CKD toggle
<FGH> - the spooler toggle

----------<ND86>----------

THE KEYBOARD TYPEAHEAD KTA -- BT=Y

The KTA allows you to input characters from the keyboard while other functions are being carried out. Note that because the DOS disables interrupts during disk I/O, speed typists will need to check that characters haven't been lost - also note that the KTA doesn't support auto-repeating keys. However, during normal operations there should be no loss of characters. BASIC has been designed to function with the KTA.

Whenever an interrupt occurs, the KTA checks to see if a new key has been pressed. If so, the KTA stores the character in a First In First Out (FIFO) buffer.

The KTA handles a request for a character in the following manner. If there is a DO file in operation, then the characters will come from the DO file and not the Type Ahead Buffer. If the spool option is enabled <^4>, then a null character is returned to the caller. Otherwise, if there is a character in the buffer, then it will be returned to the caller. If the buffer is empty or the KTA is disabled <^3>, then the keyboard is scanned in the normal manner.

The following keys control the KTA:

<^1> resets the KTA to the same status it had at boot.

<^2> toggles the View function on and off.

<^3> disables the KTA. Any characters stored in the buffer will remain there until either a <^1> in which case the buffer will be purged, or a <^3> which will cause them to be outputted in the normal fashion.

<^4> allows a spooling mode. Characters can be entered into the buffer, but no characters are returned from the keyboard until <^4> is hit again or a <^1> purges the buffer.

When in BASIC and any error occurs or if you hit the <break> or <|@> keys, the type ahead buffer is purged. When a DOS error occurs, the KTA is disabled as if you had typed <^3>. This allows you to re-enter (or edit) the command. After the command has been executed, hitting <^3> will cause the rest of the KTA buffer to be outputted. The DOS editor enables the KTA for you on exit.

The KTA is disabled by the DOS /CMD handler and remains disabled until the keyboard is scanned. This prevents, to some extent, undesirable interactions between the KTA and programs with their own keyboard drivers, e.g. word processors. The MDRET command also disables the KTA until the next keyboard scan.

----------<ND86>----------

DOS COMMAND EDITOR -- CF=Y

This routine allows you to edit the DOS commands. Simultaneous depression of the <567> keys will access the routine while at DOS or Mini-DOS ready. Under certain circumstances entry to this routine is automatic when a DOS error occurs. It is NOT entered if the error occurs under DOS CALL or if the error code is greater than 7FH or is ØØH. The CKD must be active (SYSTEM Ø CK=Y) for the routine to be accessed. The editing features are basic. Using the same nomenclature as for the CKD:

<^H>	hacks the line
<^I>	enters Insert mode
<^D>	enters Delete mode
<tab>	moves forward through line
<bckspce>	moves backwards through line
<break>	terminates Insert or Delete mode
<^break> <|break>	exits back to DOS, the command is ignored and the KTA is purged.
<^bckspce>	deletes a character
<|tab>	positions cursor at end of line
<|bckspce>	positions cursor at beginning of line
<enter>	exits to DOS and executes the command and the KTA is re-enabled. Cursor position is irrelevant.

Note that the shift backspace <|bckspce> reprints the line and re-initialises the editor.

----------<ND86>----------

BASIC SINGLE KEY ENTRY

This facility allows you to use the shifted keys, A-Z, to enter BASIC keywords. This is only in CAPS mode -- lower case mode will disable the shift key entry.
For example:-

<|L>	will enter	LOAD
<|P>	will enter	PEEK
<|U>	will enter	USING

This facility enables rapid program entry and greatly reduces the number of typing errors.

SETKEY, while at BASIC ready, allows you to change the single keystroke assignments. If no parameters follow the command, the current assignments are displayed. The syntax is:

```
SETKEY A = token [,B=token....Z=token]
```

The assignments are changed on disk and then displayed, so the write protect tab will have to be removed from the disk (if present) before changing any assignments. If no tokens are to be changed, SETKEY simply displays the current assignments. A FUNCTION CALL ERROR is returned if an attempt is made to assign an invalid keyword to a key. To check for valid keywords, see Appendix D. Just a small word of warning, be careful not to change assignments if you are using a 40 track system disk in an 80-track drive, as this could render your diskette unreadable in a 40 track drive.

QUOTE TOGGLE

Another facility in Basic key entry is an automatic case change after the quote character (") is input. This makes it easy to enter lower case characters in strings. You may turn this function off if desired -- see below.

The following SETKEY parameters give added control to your keyboard entry:

SETKEY #	disables the Single Key Entry
SETKEY $	enables the Single Key Entry
SETKEY !	disables the quote toggle
SETKEY "	enables the quote toggle

(More on SETKEY CH 12-6)

Listed below are the default keywords returned:-

A	ASC	B	RESTORE
C	CLS	D	DATA
E	ELSE	F	FOR
G	GOTO	H	GOSUB
I	INPUT	J	RETURN
K	INKEY$	L	LOAD
M	MID$	N	NEXT
O	POKE	P	PEEK
Q	CHR$	R	READ
S	STRING$	T	THEN
U	USING	V	VARPTR
W	CMD	X	RIGHT$
Y	INSTR	Z	LEFT$

-----------<ND86>----------

CHAPTER SIX

CUSTOM BASIC FOR NEWDOS86 USERS

INTRODUCTION

This package grew out of CUSTOM BASIC for '8Ø USERS, which was designed to run under most DOS's or even under a 16K tape system. Requests were made to incorporate this into the DOS itself. NEWDOS8Ø v2 was chosen as the host DOS as it offered the greatest versatility for improvements.

Some general notes about the changes to CUSTOM BASIC (CB) and to NEWDOS8Ø. The changes to CB involve mainly the concept of where the routines reside. In the original CB, the routines resided in high memory. In this version of CB, called NEWDOS86 CUSTOM BASIC (NCB), the routines reside in low memory under BASIC's buffers in two new overlay areas.

While this reduces the space available for BASIC programs, it greatly enhances the BASIC programming statements available to the user.

Since the CUSTOM BASIC routines now have tokenised keywords, previous CB users will have to convert their present CB files using the CONVERT/CMD utility. LOAD your program from Basic ready, then type:

CMD"CONVERT"<enter>

This will do the necessary conversion. A '*' is flashed to indicate that the routine has not crashed, as it takes a long time to process a large program. Normal (unenhanced) Newdos-compatible Basic programs will run under NCB and do not require conversion. This enhanced Basic will run all standard Newdos8Ø Basic programs.

Some changes have also been made to the standard ND BASIC enhancements (CBASIC).

1) Due to the fact that an enhanced line editor is now available in BASIC, the original paging provided by ND8Ø has been replaced with a new paging process.

2) The maximum length of BASIC program lines either accepted from the keyboard, or from an ASCII file, is now 255 characters.

3) The standard BASIC line editor features are still available, even if using the Enhanced Line Editor. Additionally, pressing the W key will cause the current line to be reprinted and, if possible, the cursor will be positioned close to the cause of the error. This is discussed more fully in the section on the E. L. Editor.

4) In addition to A,D,E and L as shorthand commands, R = Run, S = Save and K = Kill have also been added. Additionally, the requirements regarding the character/s following these letters have been changed. If an '=' follows the character/s, the string is assumed to be a variable. '.', '-','"' or null are allowable characters to follow the letters. These characters perform the same function as if you had typed the shorthand commands in full.

5) L? is now accepted as a shorthand for LPRINT.

6) Hexadecimal values are now valid on the DOS command line invoking BASIC. Don't forget to add a leading zero if the first Hex digit is in the range A-F. E.g.:

```
BASIC 3,ØFCØØH
```

7) Another two parameters have been introduced to the BASIC invocation. These two parameters must be specified immediately following BASIC -- before specifying the number of buffers required or the Himem setting.

The first parameter is the minus '-': this inhibits the ELE and you only have Level 2 editing. Thus BASIC,- <ENTER>.

The second parameter is the '!'. This allows you to reserve space below BASIC's buffers for your uses (i.e. move Basic buffers and program (PST) up by the specified amount). E.g.:

```
BASIC,!3ØØ,3,ØFCØØH   or   BASIC !12CH,3,ØFCØØH
```

This has the advantage in that you will always be able to ORG your USR routine(s) at a known safe spot in memory, regardless of the number of buffers you specify or the high memory setting. See the section on USR for more details (Ch 6-2).

Some general comments about using the new features of BASIC. All parameters, unless otherwise noted, can be any valid BASIC expression. This means that they can be variables or literals or a combination of both. For example:-

4*A, 6+4,"ABC",A$+B$

are all valid expressions. In this documentation "exp" indicates an expression while "var" indicates a variable. If var is specified, then expressions are not allowed. Note too, that optional parameters are enclosed in square brackets [...].

Unless otherwise noted, most CUSTOM BASIC commands requiring numeric parameters will accept values in the range plus/minus 93ØØØ before any OVERFLOW errors are reported.

EXITING BASIC

To return to ND86 Ready from Basic, CMD"S has been abbreviated to plain CMD. This can be entered by <Shift-W><enter> in the caps mode. However, CMD"S still works if entered. If in the ELE, don't forget to press <Shift-Break> before the <Shift-W> to ensure that it's written to a valid part of the screen.

REDUNDANT AND SHIFTED COMMANDS

Commands and facilities in CUSTOM BASIC FOR '8Ø USERS no longer supported by Newdos86 (because of redundancy away from the tape situation) are:

```
BUZZ
KEYBOARD SOUND
LOAD
LOAD SYSTEM
MERGE
SAVE
$SORT     use NewDos8Ø's CMD"O"
```

Commands and facilities that have been relegated back to DOS:

```
DUMP           -> CMD"JKL
LPRINT LEN =   -> FORMS"parameters"
LPRINT SET     -> CMD"route,do,do,pr"
LPRINT RESET   -> CMD"route,do,do"
SPOOLER        -> CMD"FORMS SPOOLER=xxxx"
```

----------<ND86>----------

CHAPTER SEVEN

ENHANCED LINE EDITING

GETTING STARTED

Since the Enhanced Line Editor is a very simple 'word processor', I shall try to lead you through a simple editing session. Enter BASIC and the following first screen will be displayed if the Control Key Adaptor isn't installed:

```
Disk Basic. Radio Shack's ROM enhanced
With Apparat's NewDos8Ø extended basic and disk features.
Further enhanced by

                    CUSTOM BASIC
                         For
                      '8Ø USERS

Especially adapted for NEWDOS86
     By W. S. and D. S. Sands

                Enhanced Line Editor not available
                Set SYSTEM Ø CK=Y  and re-boot
                Or hit  < enter >  to  continue
```

If you decide to hit <enter>, the BASIC prompt will revert to the '>' and you will have the standard Level 2 editing capability.

Since we're now interested in the ELE, we will have the CK adaptor installed and the following first screen will be displayed:

```
Disk Basic. Radio Shack's ROM enhanced
With Apparat's NEWDOS8Ø extended basic and disk features.
Further enhanced by

                    CUSTOM  BASIC
                         For
                      '8Ø USERS

Especially adapted for NEWDOS86
     By W. S. and D. S. Sands

±
```

The first thing you have to realise is that you have to re-orient your thinking about line editing -- as ELE is a completely different approach -- and that you will have to be patient and have a lot of practice initially until the process is familiar, after which you will be 'hooked' and find editing thereafter "a piece of cake". A habit that must be acquired is to ALWAYS PRESS SHIFT-BREAK before typing a direct command when the ELE is installed.

On initialisation, hitting any key will cause the screen to clear and you will then be able to enter BASIC commands. The secret in using this ELE is to consider the screen to be a blank piece of paper on which you are going to write your commands.

CONTROL KEY

We now are going to be using a <control> key. Models I and III and System 8Ø (Video Genie) will use the <CLEAR> key. Models 4 & 4P will use either the <CTRL> or <CLEAR> key. For brevity, <control> will be represented hereafter by <^> and <shift> by <|>. Since the arrow keys are used for cursor movement, the exponent symbol, [, is obtained by the <^,> combination. A full list of the keys and their functions is included at the end of this section.

The first thing to try is to find out how big the 'page' is. Use the <arrow> keys to drive the cursor around the 'page'. Notice that the top line of the screen is a 'forbidden' area. This line is reserved for error messages and various DOS functions such as the CLOCK display and the Keyboard Type Ahead display. Obviously, you don't want BASIC to think that the CLOCK display is some kind of command statement, so it isn't allowed on the 'page'.

Notice the character in the top right hand corner of the screen? In the Model I it will be a 'honeycomb', while in the models III and 4 it is the combined plus/minus sign and is represented in this text by '±'. This is both the end of line marker (eol) and also the prompt indicating that BASIC is ready for you to enter a command. Hit <enter> a few times. Under normal BASIC editing, this would give you a series of '>' characters down the screen. Not so with the ELE. If the last non-blank character is an eol, then the prompt is not reprinted.

The eol is used by the ELE to determine where one line of BASIC ends and another begins. You can enter it yourself with the <^|enter> combination.

Hit <|uparr>, this takes you to the 'top' of the page. <|clear> will then clear the page for you. Type in some characters on the top line, go to the bottom of the page and continue pressing the <down arr>, notice that the characters you typed at the top of the page have now scrolled off the screen. Those characters have gone forever once they're off the screen!

You can't get them back because the only place those letters existed was on the screen! This is the major difference between this 'word processor' and the standard types such as SCRIPSIT. Let's have a brief look as to why this is so.

For text editors such as SCRIPSIT, the 'page' is memory while the screen is simply a 'window' looking at your text. As you scroll through your text, it is copied from memory to the screen. As you edit your text, the text is changed in memory and the changes made are reflected in the screen display.

This BASIC editor considers the screen itself to be the page. Nothing is stored (changed) in (program) memory until you hit <enter>. When this happens, the relevant part of the screen page is copied into a buffer, tokenised and either executed as a direct command or stored in memory as a program line.

There is a problem using this technique and it occurs whenever text is printed to the screen. Under these circumstances the screen will contain 'invalid' information. Either clear the screen or use <^@> or <^a> to reprint the previous screen. This problem can also occur if you break a program, with text information on the screen.

Try it. Clear the screen and type:

? "FRED" <enter>
the screen will look something like this:

```
? "FRED"+
 FRED

+
```

with the cursor flashing after the last '+'. Hit <enter>. Notice the cursor moves up one line, to below FRED. If you hit <enter> again, you will get a syntax error. For now FRED is considered to be a BASIC command that you have inputted.

SINGLE KEYWORD ENTRY

We'll now look at the single keyword entry routine. Make certain that you can only type capitals, <|Ø> toggles the caps lock (also the <CAPS> key in the 4/4P models). Hold the <|> key down and press the letters A-Z. Notice that they cause various BASIC reserved words to appear. Go to the top of the page and clear the screen <|^>. Now type in SETKEY <enter>, this displays the current key assignments for you. Now type in SETKEY A=AUTO, B=WHILE <enter>. An ?FC error results to tell that WHILE is not a LEVEL 2 reserved word. Notice that 'A' now returns AUTO, while 'B' still returns RESTORE. Later you can alter these assignments to suit your own preferences (out of the list of reserved words in APPENDIX D-1). Also see CH 5-5.

The single keyword feature only works while the CAPS lock flag is active. When you are typing lower case characters, the routine is disabled. Every time you type a double quote, the caps lock status is automatically toggled. This makes it easier to enter lower case text in PRINT statements. This feature can be disabled if required by SETKEY! <ENTER> and re-enabled by SETKEY" <ENTER>.

Let's now examine a BASIC program with the ELE. Clear the page and type LOAD"PATCH/BAS <enter>. Now hit <^@> or <^a>. The first 'page' of the program will now be displayed on the screen. Now page down through the program with the <|downarr> key combination. Notice how the last line of the previous screen becomes the first line of the current screen? You can page back through the program with the <|uparr> key combination.

At times this may cause a problem in that you can't get beyond a particular line, this is because only that line will fit on the screen. The <^downarr> key will display the next program line and cause the top line to be lost. The <^uparr> key will let us scroll up through the BASIC program one program line at a time. This is approximately the customary ND action, except that it's much faster and the lines are always displayed in their proper order.

Now try the <|bckspce> and <|tab> keys. Notice that the <|tab> takes you to the last non-blank character of the line and the <|bckspce> takes you to the beginning of the line. When the final line on the screen is reached it forces the next program line to be displayed, like the <^downarr>.

Hit the <^|downarr>, this prints the last page of the program. Hit <^|uparr> -- this prints the first page of the program.

Spend some time in becoming totally at ease using these keys and that time will be a wise investment in retraining you to the ELE approach.

Now let's try to do some editing. I have endeavoured to keep the keystrokes required to perform a function the same as in the BASIC line editor, the main difference is that the <^> must also be pressed. Nevertheless, keep a list of the function keys handy and refer to it often. You are normally in an overtype mode. You can go into an insert mode <^i> or a delete mode <^d>. When you change mode, the cursor character changes.

Get the first program page on the screen <^|uparr>.

```
 5 CLEAR 1ØØØØ:DEFINT A-Z:DIM OL$(16):DA%=4Ø+
 6 GOTO1Ø+
 7 SAVE"PATCH/BAS":END+
 8 (12)NM$,ST%,KN$;+
 9 (16)NL$;+
1Ø CLS:CL$=CHR$(31)+
9Ø FIELD#1,16 AS OL$(1),16 AS OL$(2),16 AS OL$(3),16 AS OL$(4),1
6 AS OL$(5),16 AS OL$(6),16 AS OL$(7),16 AS OL$(8),16 AS OL$(9),
16 AS OL$(1Ø),16 AS OL$(11),16 AS OL$(12),16 AS OL$(13),16 AS OL
$(14),16 AS OL$(15),16 AS OL$(16)+
```

Move the cursor down to line 1Ø and change the 1Ø to 2Ø. Don't hit <enter>, instead hit <^@>. The screen is reprinted exactly as before. Try it again. This time hit <enter> before you hit <^@>.

Note that line 2Ø is a duplicate of line 1Ø. Position the cursor after the line number in line 6. Hit <|clear>. Notice that the screen is cleared from the cursor only. Hit <enter>, then <^@>. Note that line 6 is still in the program. Lines can now only be deleted by the direct DELETE command (Dline# or Dline#1-line#2).

It is possible to delete text. To do this hit <^d> to enter delete mode. Notice that the cursor has changed to a '\'. To combine lines, position the cursor over the eol marker at the end of line 5. Hit the <^tab> key until you have deleted to the code you wish to retain. Hit the <tab> key without holding down the <^> key. Notice that the cursor position has not changed, but the cursor character has reverted to normal? That <tab> took you out of delete mode. In fact, as soon as you hit any key without the <^> key down, you exit delete mode. Hit <enter>. Notice that the cursor moves down to the next line, indicating that the line has been stored. Now hit <^@>. You should see:

```
5 CLEAR 1ØØØØ:DEFINT A-Z:DIM OL$(16):DA%=4Ø:GOTO1Ø+
6 GOTO1Ø+
7 SAVE"PATCH/BAS":END+
etc
```

Notice that line 6 is still in the program.

To insert text at the cursor position, hit <^i>. Notice that the cursor changes to the underline chr '_'. It only requires a single <enter> to store the line, since it is always possible to delete newly inserted text, but once the line is stored, the deleted material is gone forever. To create a blank line in the middle of the screen when in insert mode use <^downarr>.

As you can see, editing a BASIC program is very simple. I find that I often use the <^@> key to reprint the page and the <|clear> to tidy up the page.

Using the screen as a 'page' can cause problems when editing some programs. These are programs that have machine code either embedded in strings or hidden behind REM statements. These can easily be fixed using the CONVERT utility -- see the section on USR...DEFUSR (Ch 9-17). The other problem occurs when the program line is longer than 15 video lines. The only solution to this problem is to either rewrite the line or use the standard BASIC line editor for any editing that might need to be done. So, in fact, there's an easy cure to such problems and, fortunately, both situations are relatively rare.

Before we start discussing advantages in writing programs, we will look at the way the ELE handles spaces displayed on the screen. This is necessary because there is no way in which we can store a CHR$(1Ø) in the screen memory -- unless we 'steal' another character to symbolise it, which I am unwilling to do.

SPACES:
How does the ELE treat spaces? Leaving text between double quotes aside, a CHR$(9) is substituted whenever 4 contiguous spaces are encountered or a CHR$(1Ø) if the rest of the video line is blank. This means that you can indent your programs like PASCAL structuring without having to worry unduly about program space limitations.

The spaces between the quotes raise all sorts of problems. We can't simply substitute a CHR$(1Ø) whenever the end of the video line is encountered, because many programs written for 132 character printers may have the printer text spanning lines. Yet if we simply store the spaces as is, we could end up with lines 1ØØØ characters long, of which 95Ø bytes are the spaces at the end of the video lines.

The ELE takes the following approach. If you hit <enter> the line is stored and all characters between quotes are stored as is, e.g. line 12ØØ in PATCH/BAS:

```
12ØØ CLS:DEFINT A-Z:CL$=CHR$(31):PRINT"


                        The SYSTEM files have now been patched.

                        Now it is necessary to set up the
                             SYSTEM options to suit your requirements."+
```

would make a line length of about 43Ø bytes.

Unfortunately, for various reasons including that of compatibility with other DOS's, we can't have lines longer than 255 characters. Lines longer than 255 bytes causes the top line to become CHR$(191)'s. This indicates that the line is too long, and was chosen over an error message in that it is somewhat more visible. The <enter> key is ignored, the cursor is positioned in the line close to the maximum line length allowed (This gives you the chance of reducing the length of the line).

Not a good situation for full screen displays. To overcome this problem you can also store a line by hitting <^G>, but now all the spaces inside the quotes have been converted into either CHR$(1Ø)'s or space compression codes. Thus the line length will now be much less than 2ØØ characters.

Now for some thoughts on writing programs, as against editing them. Setting up full screen menus are now very simple. Position the cursor at the top of the screen, <|up arr>, then clear the screen. <|clear>. Then type the line number, PRINT@ statement, and double quote (") move the cursor down one line and then simply drive the cursor around the screen putting text where you want it to go. When it meets your satisfaction, hit <^G> and the line will be stored away.

A simple example:

```
1ØØ CLS:PRINT@Ø,"
       What do you desire to patch ?

            1)    The DOS SYSTEM files
            2)    EDAS/CMD  (ver 4.1)
            3)    FEDII/CMD
            4)    SUPERZAP/CMD
            5)    Exit the program

                        Enter your choice ";CHR$(14);+
11Ø DO:DO:A$=INKEY$:UNTILA$>"":A=ASC(A$)-48:UNTIL(A>Ø AND A<6)+
```

As mentioned earlier, you can indent your programs a little to make the flow somewhat easier to follow. To be honest, personally I don't normally bother, but some people find it helps in debugging.

```
2Ø1Ø CASE -1+
2Ø2Ø ;(ASC(OL$(2))=3) 'Model I+
2Ø3Ø    IF (PEEK(1453)=Ø) AND (PEEK(12289)<>92) THEN::
        ELSE MID$(OL$(1),1Ø)=STRING$(4,Ø)+CHR$(62)+CHR$(1)+
2Ø4Ø ;(ASC(OL$(2))=211)  'Model III+
2Ø5Ø    IF (PEEK(12329)=Ø) AND (PEEK(68)>Ø) THEN
            MID$(OL$(2),1)=CHR$(1):MID$(OL$(2),6)=CHR$(144)+
2Ø6Ø ENDCASE+
```

where you see 4 spaces, there is only 1 CHR$(9), and where there are 8 spaces there are 2 CHR$(9)'s in the Basic Program Statement Table (PST).

When in Auto mode, if it is desired to skip from line 1ØØ to line 1ØØØ, just move the cursor over to the line number and insert the extra 'Ø'.

To duplicate a line in memory, just edit the line number to suit and then hit <enter>.

THE EDIT CONTROL KEYS

<Control> is represented by <^> and <shift> by <|>. Since the arrow keys are used for cursor movement, the exponent symbol, [, is obtained by the <^,> combination. Please note that the ELE will not function in 32-character mode. Either <^@> or <|clear> will cause the screen to revert to 64-character mode if 32-character mode is entered.

<^|up arrow>	prints the first page of the program.
<^|down arr>	prints the last page of the program.
<|up arrow>	prints previous page of program.
<|down arr	prints the next page of program.
<^@> <^a>	reprints the current page from memory.

<up arrow>	moves the cursor up the screen.
<down arr>	moves the cursor down the screen.
<|backspace>	to start of current video line.
<|tab>	to end of current video line.
<tab>	moves cursor forward through text.
<backspace>	moves cursor back through text.
<|^>	erases to end of screen. Use it often.
<^tab>	prints spaces to the next TAB position.
<^backspace>	deletes previous character.
<enter>	stores the current line as displayed.
<break>	ends Insert and Delete modes.
<|break>	exits BASIC auto mode without entering line.
<^g>	stores the current line but ALL spaces inside quotes are converted to CHR$(1Ø)'s or space compression codes.
<^h>	(Hack) deletes from cursor to end of current line.
<^i>	set insert mode.
<^d>	enter delete mode, then subsequently delete one character at a time.
<^e>	edits current line (same as old ND8Ø',').
<^:>	edits last entered command.
<^c>	allows an ASCII chr to be entered.
<^s>	search for specified chr.
<^k>	kill to specified chr.
<^space>	is used to begin the numeric entry of the value for Search or Kill and also to create an ASCII character.
<^up arrow>	moves up one line. If top of screen, previous BASIC line displayed.
<^down arr>	moves down one line. If at end of screen next BASIC line displayed.

INSERT MODE

<^down arr>	Inserts a blank line into the text.

DELETE MODE

<^down arr>	Deletes a line.
<^tab>	Deletes one character and all blanks to the next non-blank character.
<^d>	Deletes one character.

To enter the numeric value for Kill and Search use the <^space> key. Hit <^space>, the number desired and then the function desired. Thus <^space> 191 <^C> would cause the full graphic block to be entered at the current cursor location. And <^space> <2> <^S> : would search through the program line for the second colon.

Hitting any key, other than a number key, takes you out of numeric entry. <^space> resets the current value to zero. The routine won't allow the value entered to be greater than 255. The current value is displayed at the top of the screen.

Some comments regarding the <^E> routine in the E.L.E. and the <W> command in the standard BASIC line editor:

<W> causes the line to be listed as if <L> was hit and then prints the line up to the point where the error occurred.
<^E> clears the screen, prints the line in full and then positions the cursor over the statement in question (if this is possible).

The error should be pinpointed to within a particular statement, in many cases to the character in question. The main exceptions involve errors within a Function since the needed information isn't available.

----------<ND86>----------

=============== NEWDOS86 (C) 1986 W.S. & D.S. SANDS ===============

CHAPTER EIGHT

BASIC UTILITIES

DI - DU

The standard NewDos8Ø DI-DU command has been extended to enable the moving of blocks of program lines as against the single line move of the original DI - DU commands.

```
DI line1 [-line2], line3
DU line1 [-line2], line3
```

line1 is the first line to move.
line2 is the last line to move [optional].
line3 is the line to move the code to.

As an example:-

```
1Ø      CLS
2Ø      A=1Ø
3Ø      B=12
4Ø      C=15

DI 1Ø-2Ø,3Ø     <enter>
LIST    <enter>

3Ø      B=12
3Ø      CLS
2Ø      A=1Ø
4Ø      C=15
```

Automatic renumbering is not done, as there may be times when you want to have out-of-sequence line numbers. If you want automatic renumbering to occur, use the NewDos8Ø RENUM to move the lines. The CUSTOM BASIC REN utility (see CH 8-3) will correctly renumber all lines that have been moved by DI-DU.

Note that the original line 3Ø has not been deleted, thus preventing its inadvertent loss. D3Ø will delete the original line 3Ø if required. In the above example, line 2Ø can no longer be found by the ROM. Line 2Ø will be executed after both line 3Ø's have been executed, but a GOTO 2Ø will result in an UNDEFINED LINE ERROR. If we only wish to RENumber the moved lines (so as to retain the previous line number structure) we need to 'bracket' the moved lines with a pair of dummy lines so the ROM can find them:-

```
1Ø      CLS
2Ø      A=1Ø
3Ø      B=12
31      REMOVE THIS LINE AFTER RENUMBERING
32      REMEMBER TO REMOVE THIS LINE AS WELL
4Ø      C=15
```

```
DI 1Ø-2Ø,31     <enter>
LIST     <enter>
3Ø       B=12
31       REMOVE THIS LINE AFTER RENUMBERING
31       CLS
2Ø       A=1Ø
32       REMEMBER TO REMOVE THIS LINE AS WELL
4Ø       C=15
```

Line 2Ø can now be accessed by referencing the REM line 31. To RENumber the original lines 1Ø and 2Ø without affecting the rest of the program the following statement can be used:-

```
REN 31,2,31,32     <enter>
LIST     <enter>

3Ø       B=12
31       REMOVE THIS LINE AFTER RENUMBERING
33       CLS
35       A=1Ø
37       REMEMBER TO REMOVE THIS LINE AS WELL
4Ø       C=15
```

Lines 31 and 37 should now be deleted.

BRANCHES It has been stated in many computing publications that all program subroutines should be at the start of the program. This is not always the case.

Before BASIC starts the line search to find the line referenced by a GOTO or a GOSUB call, it first checks the relative positions of the calling and called line. If the GOSUB'ed line is before the calling line, the search is begun at the start of the program, otherwise the search is started from the calling line.

So, if you want your program to RUN faster, organise your program so that GOSUB's common to the entire program are at the start of the program. GOSUB's that are referenced rarely, should be at the end of the program. GOSUB's that are referenced often, but only from a certain area of program, are best placed at the end of the block of code from which they are called.

If you have nested GOSUB's (GOSUB's that do a GOSUB that do a GOSUB), it's best to try to organise them so that the third GOSUB has a greater line number than the second GOSUB which has a greater line number than the first GOSUB. Usually this is the best method.

The search mechanism takes approximately 15Ø microseconds per line (Model I). Obviously this will vary according to your machine's clock rate. That means that it takes about one second to skip 6ØØØ lines of program. So, if speed is a matter of concern, organise your program so that the minimum number of lines have to be skipped.

----------<ND86>----------

RENUMBER

Even though NewDos80 has an excellent renumber routine, the CUSTOM BASIC routine has been retained as well. This renumber utility handles programs with the line number 0. Additionally, if there is a reference to a non-existent line number, that reference is not changed and no error is flagged. Use the NewDos80 RENUM U to check for unresolved references. The format for the RENumber routine is:-

REN [starting value], increment], start], end]

all values must be literal strings. The RENumber utility has several options:-

REN RENumbers the whole program, the starting line defaults to 100 and the increment is 10.

REN 50 RENumbers the whole program, the starting line has the value 50, with the increment value being 10.

REN 50,5 RENumbers the whole program, the first line is 50 and the increment is 5.

REN 50,10,200 RENumbers the program, starting with line 200. Thus the present line 200, becomes line 50, and the succeeding line numbers are incremented by 10.

REN 50,10,200,250 RENumbers the program between lines 200 and 250. Line 200 becomes line 50, with successive lines being incremented by 10. The line following current line 250 retains its present number.

If an increment of 0 is specified, a DIVIDE BY ZERO ERROR is reported, and if the renumbering process would cause the line numbers to exceed 65534 a BAD SUBSCRIPT ERROR is given. If a BAD SUBCRIPT ERROR is reported, choose a smaller increment or a lower starting value or both. The error is reported before any renumbering is actually done, which ensures that the program will remain intact. Even though the lines may be inaccessible by the ROM, they will still be RENumbered successfully. See DI-DU

The RENumber utility checks for all the tokens listed below. Lists of line numbers are catered for (providing a comma is used as the separator) allowing ON exp GOTO 1,2,3 to be processed correctly. Since we test for tokens such as RESTORE, always have the offset (not a variable) as the first parameter following the #RESTORE. Then when a RENumber occurs, the base address will be adjusted so that it is still referencing the correct line. As an exercise, type in the program in Ch 9-10 and try RENumbering it with various parameters and note the result.

GOTO,EDIT,LIST,LLIST,GOSUB,THEN,ELSE,RUN,RESTORE
RESUME,DELETE,LOAD,MERGE,SAVE

-----------<ND86>----------

VAR[L] [*] [#] namelist

This command will cause an output of all of the simple variables, their contents and type to be sent to the monitor.

The options include:-

L is suffixed to the VAR command then the output will go the printer instead of the screen.

* indicates to print only the array variables.

# indicates to print out only the simple variables.

The name list only prints variables which fall in the specified range.

Since the values within an array variable can be printed out within a FOR-NEXT loop, VAR merely indicates the dimensioning of the array. VAR pauses at the end of each screenful, hit <enter> to move onto the next screen.

The time required to find a variable is a function of its position in the table. Since the interpreter must search the variable table each time a variable needs to be accessed, often accessed variables should be the first ones used in a program.

As the VAR routine works down the variable table in the same manner as the BASIC interpreter, the position of the variables in the table can be easily determined from their position in the listing.

The DIM command can be used to initialise your variables in the desired order of occurrence, so can be changed after studying the VAR listing, to improve the execution speed of your program.

An example:-

```
VAR a-c,m     <enter>

 A $          JOE BLOGGS
 AB%           32456
 A #          -123
 B $          A funny little program
 M %          12
 CN$          User Defined Function
 CA$(12,12)
```

----------<ND86>----------

CHAPTER NINE

CUSTOM BASIC COMMANDS

BASE CONVERSIONS

&%, &!, &H, &D, &O and &B

&O and &B allow the conversion of OCTAL and BINARY numbers to decimal. Also the &H, &O and &B functions will accept string expressions as arguments if these are enclosed in parenthesis. E.g.:

```
        PRINT &H (A$)
```

is now a valid command. &! forces the result of an expression to be a single precision number in the range 0 to 65535. Thus:

```
        PRINT &H F000, &!&HF000
```

would give

```
       -4096     61460
```

&% forces its argument to within integer range. Thus:

```
        PRINT &%61460
```

would give

```
       -4096
```

DECIMAL to HEX conversions may use expressions, the result of &D exp is a five-character string, ending in H. For example:-

```
        PRINT &D 15
```

will return as the answer:

```
        000FH
```

Try out the following program to examine memory locations from X to Z:-

```
FOR I= X TO Z:B=PEEK(&%I):PRINT &DI,B,&DB,
     CHR$(B*(B>32)*(B<192)):NEXT
```

If the leading zeros and trailing 'H's become tiresome, try:-

```
FOR I= X TO Z:B=PEEK(&%I):PRINT LEFT$(&DI,4),B,
       MID$(&DB,3,2),CHR$(B*(B>32)*(B<192)):
       NEXT
```

----------<ND86>----------

TONE GENERATION

BEEP tone,duration or BEEP exp$

BEEP 23,30: BEEP STRING$(50,50)

This function delivers an output signal via the cassette port and thus requires an audio output device to be connected to this port.

The greater the value of 'tone', the higher the note delivered, while the length of time the tone is produced, is determined by the value of 'duration'.

The limits of the expressions involved are 0 - 255. Values out of this range produce a ILLEGAL FUNCTION CALL error.

To allow short "tunes" to be produced, the BEEP command can be used with strings which contain the tone-duration pairs. Thus:-

BEEP CHR$(20)+CHR$(40)+CHR$(10)+CHR$(30) <<enter>>

is the same as:-

BEEP 20,40:BEEP 10,30 <<enter>>

This command would be useful in debugging BASIC programs as it can give an audible confirmation that a particular line was encountered without halting the program and without destroying your screen layout. This feature is a good alternative to the STOP command.

----------<ND86>----------

=============== NEWDOS86 (C) 1986 W.S. & D.S. SANDS ===============

CALL address USING var%x

```
CALL &H 1C9 USING A%(Ø)
CALL DEFUSR 8 USING IX%(Ø)
```

This allows you to call a machine code program and pass to it and receive from it all the normal Z8Ø registers. The format of the INTEGER array is as follows:

A%(Ø)	HL
A%(1)	DE
A%(2)	BC
A%(3)	AF
A%(4)	IX
A%(5)	IY
A%(6)	HL'
A%(7)	DE'
A%(8)	BC'
A%(9)	AF'

The values in the integer array are passed into the various registers and then the routine at the specified address is called. When the routine RETurns, the contents of the registers are loaded back into the integer array allowing you to examine the result of the routine called.

!!!!WARNING!!!! IF YOU DON'T KNOW WHAT YOU !!!!WARNING!!!!
!!!!WARNING!!!! ARE DOING -- DON'T USE CALL !!!!WARNING!!!!
!!!!SEEK EXPERT ADVICE FIRST!!!!

-----------<ND86>-----------

CHR$64 Takes the video out of 32/4Ø character mode without clearing the screen.

-----------<ND86>-----------

DEC var% Equivalent to var%=var%-1. Decrease the variable's value by 1. Must be an integer variable.

-----------<ND86>-----------

USER DEFINED FUNCTIONS

DEF FN, FN

```
DEF FN  A$ = A$+"#"
A = FN X$(X,Y)
```

CUSTOM BASIC allows for the use of user defined functions. This feature can save much programming time and complements the #DO command.

The advantage of using the FN instruction rather than simply doing a GOSUB to a line of code, is that the FN instruction may be used within IF...THEN...ELSE commands, without a variable being needed to flag the result to the calling routine.

Functions may be nested, but to avoid ambiguity, it is advisable to use variable names in the argument lists that are different from those in the function definitions. Function definitions must only occur within program lines, otherwise an ILLEGAL DIRECT ERROR occurs. Any errors within the function definition are reported in the line which calls the function.

The format for this command is:-

DEF FN var [(parameter list)] = function definition

Var1 = FN var [(argument list)]

The function definition may be any mathematical, or string, formula. Direct BASIC commands, such as GOTO, RESTORE or IF...THEN...ELSE are not allowable.

The parameter list is optional, but can only include variables, while the argument list may contain expressions. If variables are specified in the parameter list, they are not affected when the function is called.

The only limitation involved is due to the method of preserving the string variables. If the total number of string variables (in the parameter list) plus string operations (in the function definition) exceeds eleven, a STRING FORMULA TOO COMPLEX ERROR may be generated. Within these limitations, the function may be as complex, or as simple as you desire.

The location, within the program, of the function definition is stored in the variable table, and is accessed by finding the variable name specified. To allow the BASIC interpreter to distinguish between functions and variables, the function name has the seventh bit of the first character set.

An example: To determine the circumference of a circle the following formula is used:-

Circumference = 2 * PI * Radius

This can be defined as a function within a program:-

```
 1Ø   DEF FN CI(A) = 2 * 3.14159 * A
 2Ø   PRINT FN CI(4)

 RUN  <<enter>>

 25.1327
```

Once the function has been encountered it may be used as would any other variable. Thus:-

```
PRINT FN CI(X)  <<enter>>
```

will display the circumference of a circle with a radius equal to the value of the variable X.

A slightly more complex function definition using logical operations: SET/RESET/POINT and #DRAWc demand values in the range Ø<=X<128 AND Ø<=Y<48. A function can be defined which will prevent the values from exceeding these limits:-

```
DEF FN XY(A,B,C) = A-((B<Ø)*(A>Ø))+((B>Ø)*(A<C))
```

where A can be the X or Y co-ordinate

```
If B < Ø,  A = A-1
If B = Ø,  A=A
If B > Ø,  A=A+1
   C =  the maximum co-ordinate value + 1
```

To increase the X co-ordinate:-

```
X = FN XY(X,1,128)
```

To decrease the Y co-ordinate:-

```
Y = FN XY(Y,-1,48)
```

Obviously this function will be useful wherever you desire to keep a value within a certain range.

An example using strings: A program requires delimiters to be inserted between strings. We could DEFFN the function as follows:-

```
10   DEF FN A$(B$,C$,D$,E$) = B$+E$+C$+E$+D$
20   K$="xyz":A=123
30   M$ = FN A$(K$,"ABC",STR$(A),CHR$(64))
40   PRINT M$

RUN  <<enter>>

xyz@ABC@ 123
```

As with almost everything the decision to use functions, GOSUB's or just repeat the code is a trade off between speed of program execution and memory size. But the DEF FN is a powerful weapon in the arsenal of the programmer. Learn to use it well, and it will save you much programming time. Below are some functions that you will find useful. (Also see Lewis Rosenfelder's book 'BASIC FASTER and BETTER' for more examples.)

CONVERTS RADIANS TO DEGREES

```
DEF FN RD(A) = A*57.29578
```

CONVERTS DEGREES TO RADIANS

```
DEF FN DR(A) = A*0.0174533
```

RETURNS INVERSE SIN or ARCSIN

```
DEF FN AS(A
 = 2*ATN(A/1+SQR(1-A*A)))
```

RETURNS INVERSE COS or ARCCOS

```
DEF FN AC(A) = 1.5708 - FN AS(A)
```

RETURNS THE SMALLER OF TWO NUMBERS

```
DEF FN MN(A,B) = (A+B-ABS(A-B))/2
```

RETURNS THE GREATER OF TWO NUMBERS

```
DEF FN MX(A,B) = (A+B+ABS(A-B))/2
```

RETURNS THE REMAINDER OF A/B

```
DEF FN MD(A,B) = A - B * INT(A/B)
```

Don't forget that the &% function can be used in any of the above function definitions and will automatically convert the result to a value within integer range.

----------<ND86>----------

DEFUSR -- See USR (CH 9-17)

----------<ND86>----------

ERROR
ERR$

This routine has been slightly expanded in that it is now possible to cause the DOS BASIC errors to be fully displayed. Originally, ERROR 51 caused an UNDEFINED ERROR to be reported. Now an ERROR 51 will give FIELD OVERFLOW.

ERR$ returns the expanded error message as a string. This allows your error handling routine to manipulate the error message as you would any other string. Thus:

```
A$ = ERR$(18):PRINT A$
NO RESUME
```

Additionally you can use other unassigned error numbers within your own program. Whilst an ERROR 126 will display as UNDEFINED ERROR, the ERL/2+1 value will be 126. So if you use an ON ERROR GOTO routine, you can check for your own internal errors within your program's ERROR handler.

```
10 ON ERROR GOTO 2000:DIM CP%(512)~
.....
300 INPUT @[4,10],"Enter account number";A$~
310 IF VAL(A$) = 0 ERROR 120~
320 AN%=VAL(A$):RETURN~
.....
2000 ER%=ERR/2+1:CP%(0)=WPEEK(&H4020) 'Save cursor Pos~
2010 #MOV 15360,VARPTR(CP%(1)),1024  'Save the screen~
....The error handler itself resides here
....It can determine from whence the error came and
....what sort of error message to display
....IF ER% = 120 then Bad account number
....or other type of error....
....at exit some finalising code
2090 PRINT @[15,30],"Hit <any key> to continue";~
2094 DO:UNTIL INKEY$>"":WPOKE &H4020,CP%(0)~
2097 #MOV VARPTR(CP%(1)),15360,1024 'Restore screen~
2098 RESUME~
4000 ON ERROR GOTO 0:END~
```

----------<ND86>----------

FCB

var = FCB (exp)

Returns the address of File Control Block specified by exp. The FCB must be in the range specified at entry to BASIC. It is not necessary for the FCB to be active, i.e. the FCB need not contain an OPENed file.

This function allows you to use BASIC's FCBs and Buffers to manipulate disk sectors directly using the standard DOS primitives by means of the CALL command, E.G. 4436H to read a sector, 4439H to write a sector.

A point regarding the layout of BASIC's FCBs. The first 13 bytes of BASIC's FCBs are used internally by BASIC itself. The following 32 bytes are the standard DOS FCB, this is followed by the buffer for the FCB. The value returned by FCB(x) is the starting address of the DOS FCB, therefore the start of BASIC's FCB is FCB(x)-13.

----------<ND86>----------

FILL

This command is similar to the #MOV -- the difference being that #MOV automatically determines whether an LDIR or LDDR is required and uses same. At times it is desired to quickly FILL or clear memory and the #MOV is not suitable for this task. The syntax is:

FILL source address, length1, target address, length2

No checks are made, so be careful. An example:

```
FILL &H1929,5,15360,1024
```

will fill the screen with READYREADY. If length1 is greater than length2 then only length1 bytes are copied up.

----------<ND86>----------

FN -- see DEFFN CH 9-4

----------<ND86>----------

EXTENDED GOTO

#GOTO line number

    #GOTO RND(X)

The #GOTO statement is similar to the standard GOTO statement except that the line to which program execution is to branch is specified by the value of a numeric expression.

Thus if the variable N has a value of 1ØØ and the statement:

#GOTO N

is encountered, then program execution will continue at line 1ØØ. A small program follows to demonstrate this feature:-

```
25ØØ  CLS
251Ø  INPUT"INPUT VALUE FOR N (1 TO 5)";N
252Ø  #GOTO 253Ø + N*1Ø
253Ø  PRINT" A NUMBER BETWEEN 1 AND 5 PLEASE":GOTO 251Ø
254Ø  PRINT"N MUST = 1":GOTO251Ø
255Ø  PRINT"N MUST = 2":GOTO251Ø
256Ø  PRINT"N MUST = 3":GOTO251Ø
257Ø  PRINT"N MUST = 4":GOTO251Ø
258Ø  PRINT"N MUST = 5":GOTO251Ø
```

No #GOSUB command has been implemented because if it is desired to #GOSUB in a program, merely assign the expression to a variable, and do a normal GOSUB to a line which has a #GOTO command.

The main use for this command is to replace statements such as:-

ON N GOTO 1Ø,2Ø,3Ø,4Ø......21Ø,22Ø

with:-
#GOTO N*1Ø

And for GOSUB's:-

```
  2ØØ  GOSUB 1ØØØØ
1ØØØØ  #GOTO N*1Ø
```

If the line identified by the expression does not exist, then an UNDEFINED LINE error will be generated.

-----------<ND86>----------

INC var%   Equivalent to var%=var%+1. Increase the variable's value by 1. Must be an integer variable.

-----------<ND86>----------

INPUT @pos,>prompt$;variable list

This allows the INPUT command to be positioned on the screen and also allows the prompt string to be any valid string expression. E.g.:

```
PRINT@454,"Reading record no: ";I;:INPUT#1,A$(I)
```

can be replaced with

```
INPUT@ [6,34],"Reading record No: "+ STR$(I); #1,A$(I)
```

If you desire the prompt string to be a variable, the following syntax is used:

```
INPUT@ 454, >var$;C$
```

Note that the '>' indicates that the variable is the prompt string to be printed and not the INPUT target string.

----------<ND86>----------

KTA exp$

Allows you to set the status of the keyboard type ahead. The valid parameters following KTA are:

```
Y  =   enable KTA
N  =   disables KTA
R  =   resets and enables KTA, View off
P  =   purges the KTA buffer
V  =   sets the View option on
```

For example:

```
KTA "RV"
   Resets the KTA and turns the view option on.
```

----------<ND86>----------

MOD

var = MOD(expl,exp2)

XOR

var = XOR(expl,exp2)

These two new functions are self explanatory. MOD() returns the remainder after an unsigned integer division, while XOR() eXclusive OR's the two expressions. Expl and exp2 must be in the range +/- 93ØØØ.

```
PRINT MOD(12,5)  ->  2
PRINT XOR(8,24)  ->  24
```

If you require a remainder function capable of handling numbers outside of the above range, see CH 9-6 (DEF FN list).

----------<ND86>----------

MOVING BLOCKS OF MEMORY

#MOV from addr1,to addr2,number of bytes

#MOV Ø,15360,256

The ability to simply and easily move blocks of memory from one location to another gives an extremely powerful tool to the programmer.

With this command it is possible to set up a change of various BASIC pointers and then copy them down to overwrite the original vectors in a single instruction.

The #MOV command can also be used to achieve partial screen scroll between certain limits, thus allowing portions of the screen to remain unaffected. Try the following program:

```
 1Ø     CLEAR 1ØØØ:DIM T$(26):CLS
 2Ø     PRINT@25,"HEADING
 3Ø     PRINT@985,"BOTTOM LINE";
 4Ø     FOR I=1 TO 26:T$(I)=STRING$(1Ø,64+I):NEXT
 5Ø     G=4:PRINT@64,"";
 6Ø     FOR I=1 TO 6:PRINT:PRINT TAB(1Ø)T$(I):NEXT
 7Ø     S=PEEK(144ØØ)
 8Ø     IF S=8 THEN GOSUB 5ØØ ELSE IF S=16 GOSUB 6ØØ
                                'use <dwn arr> or <up arr> to scroll
 9Ø     FOR I=Ø TO 3Ø:NEXT
1ØØ     GOTO 7Ø

5ØØ     IF G<26, G=G+1:#MOV 15616,15488,64Ø
51Ø     IF G<25 THEN PRINT@778,T$(G+2); ELSE PRINT@768,CHR$(3Ø);
52Ø     RETURN

6ØØ     IF G>1,G=G-1:#MOV 15488,15616,64Ø
61Ø     IF G>3 THEN PRINT@138,T$(G-3); ELSE PRINT@128,CHR$(3Ø);
62Ø     RETURN
```

This program will leave the comments at the top and bottom of the screen unaltered, while scrolling the middle portion of the screen. This program can be adapted to suit many applications. If combined with the INPUT TO...USING... routine data entry becomes very simple. PEEK'ing the keyboard memory location 144ØØ returns 8 when the <up arr> is pressed and 16 when the <dwn arr> is pressed. This technique is faster than using INKEY$ and allows for an automatic repeat function.

WARNING: Since #MOV can move large blocks of memory and no checks are made, NEVER specify the number of bytes to move as Ø (ZERO), as 65536 bytes will be moved -- with an inevitable crash.

----------<ND86>----------

PRINT @pos,

The PRINT and INPUT commands now allow for two methods of positioning the cursor. The first is the standard method where pos must lie between Ø and 1Ø23. The second method allows the cursor to be positioned according to screen co-ordinates. Thus:

PRINT@ [row,column],

the square brackets indicate to BASIC that the parameters are row and column co-ordinates. [Ø,Ø] is top left of the screen, while [15,63] is bottom right.

When 8Øx24 mode has been selected (See VID -- CH 9-2Ø), the maximum limits are extended to [23,79].

-----------<ND86>----------

SPECIFIC DATA RESTORATION

```
#RESTORE line number

   #RESTORE  N*1Ø
```

This causes the next READ statement to use the first DATA item in the line specified by the argument, if the line asked for does not exist an UNDEFINED LINE error will result.

The following small program will demonstrate the use of this instruction:-

```
1Ø      INPUT"ENTER A '1' OR A '2'";A
2Ø      IF A<1 OR A>2 THEN PRINT"ONLY A '1' OR '2'
                                WILL DO":GOTO 1Ø
3Ø      #RESTORE 5Ø+A*1Ø
4Ø      READ S$:PRINT S$
5Ø      GOTO 1Ø
6Ø      DATA "LINE 6Ø DATA"
7Ø      DATA "LINE 7Ø DATA"
```

NOTE: This feature allows you to find the location in memory of any program line, whether or not it contains a DATA statement, as shown below:-

```
#RESTORE 5Ø:A=WPEEK(16639)
```

The variable A will now point to the zero byte preceding line 5Ø (see diagram). This allows a BASIC program to locate a particular line and then modify it. This method was used to construct the DATA statements contained in EXTENDED BASIC.

Zero Byte Ending Last Line	Next line Pointer		Line Number		The program Data goes Here	Zero Byte Ending This Line
	Lsb	Msb	Lsb	Msb		
			5Ø	Ø		

----------<ND86>----------

SUMMING ARRAYS

var = SUM var(start)[=ntf][STEP value],number of summations to perform.

This allows you to sum the elements in an array. Thus:

```
A# = SUM A!(Ø),1Ø
```

will cause the elements Ø to 1Ø to be added and the result stored in the variable A#. The summation will be done in single precision however. If you were working with large figures, you would need the summation to be carried out in double precision to retain accuracy.

```
A# = A!(Ø)=#,1Ø
```

would do this. As the routine uses the arithmetic ROM routines, the speed advantage is between 4 and 1Ø times, but the major advantage is that the function can be used within user defined functions. E.g.:

```
DEF FN SA# = SUM A!(Ø),1Ø - SUM B!(Ø),1Ø
```

The STEP parameter allows you to STEP across unwanted elements. If you specified a STEP of 4 then only every fourth element would be summed. This finds particular application in the summation of multi-dimensioned arrays:

A(Ø,Ø)		A(Ø,2)		A(Ø,4)
1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	1Ø	15	2Ø	25
A(4,Ø)		A(4,2)		A(4,4)

Whilst it is convenient to think of a two-dimensioned array as the chequer board above, the elements of the array are stored in the computer in the order specified by the values inside the squares.

If we wish to sum a column, e.g. the centre column, SUM A(Ø,2),5 will give us a result. To sum a row we must use the STEP parameter. To sum the centre row, SUM A(2,Ø) STEP 5.

----------<ND86>----------

#SWP from addr1, to addr2, number of bytes

#SWP 15360, 15872, 512

The #SWP command has the same format as the #MOV command. This command swaps the memory block starting with address1 with the memory block starting with address2, the size of the blocks is set by the third parameter.

This command gives the ability to swap screens to/from memory. A simple program to do this follows:-

```
10      DIM A%(511)
20      CLS:PRINT STRING$(30,"#")
30      #SWP 15360,VARPTR(A%(0)),1024 'saves the
                                screen in A%(0) through A%(511).

40      FOR I=0 TO 1000:NEXT:CLS
50      #SWP VARPTR(A%(0)),15360,1024 'restores
                                the screen.

60      WAIT:GOTO 50
```

A point to note regarding this small program: Any type of numeric array may be used, but INTEGER arrays are recommended, because of the ease in which particular locations may be altered, merely by changing the value of the array.

For example, add:-

```
39      A%(0)=32*256+32    'will blank first two
                                characters of the screen array.
```

When the array is moved back to the screen, the first two characters will be spaces.

It is also possible to swap the contents of two similar variables, or arrays. For instance, the following program will demonstrate how to swap the first three elements with the last three elements of array A%(10):-

```
10    FOR I=0 TO 10:A%(I)=I:NEXT
20    #SWP VARPTR(A%(0)),VARPTR(A%(8)),6
30    FOR I=0 TO 10:PRINT I,A%(I):NEXT
```

The #SWP command nicely complements the #MOV command and has many uses where areas of memory must be manipulated.

----------<ND86>----------

USR
DEFUSR

Machine language subroutines are essential at times to perform tasks BASIC cannot normally perform or which it cannot perform quickly enough. As well as continuing to support the standard USR and DEFUSR routines, CUSTOM BASIC provides new ways to use these commands.

This command is to aid machine language programmers who wish to have machine code routines embedded in their BASIC programs. Normally the routine is either embedded in a string literal or hidden behind a REM statement. Both techniques involve some problems. The syntax of the command is:

```
        USR [x] ['comment] = machine code routine
```

where x is a character Ø-9. The optional comment allows you to identify what the routine is doing. It must be started with the "'" and is terminated by the "=".

When the USR x command is encountered, the address of the specified USR entry is set to the byte immediately following the '=' and then the following code is skipped until the end of line character (ØØH) is encountered. This command must be the only statement in the line.

The machine code routine is accessed in the standard manner:
A = USR x(argument).
Also CALL DEFUSR x USING A%(Ø) may be used if you need to pass in all the registers.

Note that the routine itself can be any length. It isn't limited by the standard BASIC line length of 255 characters and all characters are allowable with the exception of ØØH. There are many ways to enter such a routine into the program. Most programmers will have a favourite technique, but here are two ways that are appropriate to this new command:

a) First technique. Enter a line in the program,

```
   1Ø USR Ø=*******......******
   RUN
```

and then enter DEBUG. The first character of the first screen of DEBUG will be the first byte of your code area. This allows you to find the line easily for modification. This works well with short routines. If the routine is over 255 bytes this technique is no longer valid. But note that you will actually be modifying the code within the BASIC program itself!!

b) Technique number two involves using a program to POKE the machine code into memory, the code being stored in data statements. The method would follow the following format:

```
    10      GOSUB 1000
    20      A = DEFUSR 0         'Find routine
    30      FOR I = 0 to length of routine-1
    40      read and process the data
    50      POKE I+A, processed data
    60      NEXT
    70      CALL &H 1AF8 USING A%(0)
    80      WPOKE &H 40F9,A%(0)+1:CLEAR 50
    90      LIST
      ...
   1000    USR 0 = **********
   1001    RETURN
   1010    DATA all the relevant data
   1020    DATA following the USR line
   1030    DATA 0,0,0
```

The program will poke the machine code into the lines containing the data statements, so save the program before running it. Lines 40 & 50 will vary depending on how you have stored your machine code in the DATA statements.

Note also that the DATA lines should be the last lines in the program. If you try to access lines beyond the USR line, ?UL errors may occur. Line 70 fixes the BASIC line pointers, while line 80 fixes the variable pointers.

**** NOTE THAT WHEN THIS TECHNIQUE IS USED, IT IS ****
**** ESSENTIAL THAT THE ROUTINE IS TERMINATED ****
**** WITH THREE ZERO BYTES (00H) ****

When this line is listed, or editing is attempted, the message:

    USRx routine 'comment

will be displayed instead of the line and EDIT mode will be exited. If you wish to EDIT the line to make it longer, the command:

    EDIT line# USR

must be used. This will copy the line into BASIC's buffer and enter DEBUG. DEBUG will be in the mode described before. Edit the line. Make certain that you terminate the routine with a 00H byte and then type:

    G <enter>

to get back to BASIC READY.

If you wish to abort the editing process type:

```
        G1A19  <enter>
```

and then, from BASIC READY, type

```
        CLEAR 5Ø  <enter>
```

immediately!!!!!

If the line won't fit into the buffer, a STRING TOO LONG error is flagged. If DEBUG is entered, the display will be as noted previously. However, any editing will be occurring in the program area itself, so be careful. Or you may wish to use the utility CONVERT/CMD instead.

REN acknowledges that USR lines exist and while renumbering the USR line itself, will skip to the end of the USR line. This prevents any of the machine code in the line from being corrupted by the renumbering process.

If you already have your USR lines set up, it isn't necessary to 're-assemble' them. Included on the diskette is a file called CONVERT/CMD, which is accessed from BASIC with the command:

```
        CMD"CONVERT,U,line#,['Comment string]"
```

This file will move your routine into a buffer, prefix it with the necessary information and set it up for you to modify with DEBUG. Routines up to 4ØØH bytes long can be processed by this technique. As an example, assume a routine embedded in a string:

```
     15 A$="your code
     CMD"CONVERT,U,15  <enter>
```

DEBUG will be in the X mode and the first line of the first screen will be:

```
52ØØ  AEFF 223Ø 4124 D522 xxxx xxxx xxxx xxØØ  .."ØA$.".......
```

The 'AEFF 22' is the token sequence for the USR routine. The '3Ø' is the default USR number, change it if you wish. What you MUST do is change the byte before the first valid machine code instruction to a 3DH. E.g.:

```
52ØØ  AEFF 2239 4124 D53D xxxx xxxx xxxx xxØØ  .."9A$.=.......
```

Note that the 22H (") following A$ has been replaced with a 3DH (=) and the USR routine is now USR9. After you have finished, you simply type in G <enter> from DEBUG and the line will be stored. The utility searches from 52ØØH for the first 3DH character. All characters from this point on are then treated as the USR routine. If you single step through (C instead of G), you would see the following final result of your routine.

```
52ØØ  AEFF 2239 3Dxx xxxx xxxx xxØØ ØØØØ cccc  .."9=..........

      G6FØØ <enter>
```

will abort the CONVERT U process. This resets all the vectors that CONVERT has changed, and prints the message 'aborted'.

While on the subject of USR routines, mention should be made of the !value parameter on entry to BASIC. ND86 now uses the address 66BEH as a vector to the start of BASIC's file buffers. These normally start at 74ØØH. The ! allows you to reserve some space at 74ØØH with BASIC moving up its buffers to compensate. This means that you can store routines there -- confident that they won't be moved as you edit your BASIC programs. Likewise, you can vary the number of Buffers or Himem without having to worry about absolute addresses changing.

You can store your routine as a last line in your BASIC program, and by using DEFUSR, #MOV and CMD"F",DELETE you can move it to the 74ØØH area as part of your initialisation process, or you may prefer to POKE it in using standard techniques. Another alternative is to store your routine in a command file with the PSF attribute and load it using the CMD"filespec" command.

----------<ND86>----------

VID(exp)

If your machine has 8Øx24 capability and a suitable driver installed, VID(exp) can be used to control which video mode is active. If exp evaluates to 8Ø then 8Øx24 mode is activated otherwise 64x16 mode is set.

It is best to have a VID(exp) statement early in your program to ensure correct screen layout. This statement is valid in Model 1 BASIC, obviously though without a driver the mode will remain as 64x16.

BASIC supports SET, RESET and POINT in 8Øx24 mode. Please note that due to some ROM limitations the TAB and end of line values may not always be correct. At a future time I hope to release a patched MODELA file for the Model 4P users, and either a ROM pack or a program to cater for the Model 4 users. (Please don't hold your breath while waiting for either.)

The 8Øx24 mode is supported by the PRINT@ statement. However the format must be PRINT@[row,column].

For more information please see UTILITY 13 which discusses the 8Øx24 video driver for the Model 4/4P. A model 1 driver has not been written as yet, mainly because I don't of an 8Øx24 hardware modification for these early machines. If you have such a modification and you can supply me with necessary technical information, I will endeavour to write a driver for the model 1.

----------<ND86>----------

WAIT

WAIT [@exp,][>prompt$;][FOR exp$]

This routine waits until the specified key/s are pressed. If no keys are specified in the FOR phrase, then the routine waits until DOS returns a character from the keyboard. The key which caused the routine to exit is stored in the INKEY$ and SETKEY storage locations.

The routine converts the keyboard character to upper case for testing purposes, thus 'a' and 'A' are considered equivalent. Use only upper case letters in the FOR string, the routine ignores lower case letters. However the character returned by INKEY$ is not converted to upper case. Examples:

```
WAIT
WAIT "Hit <any key> to continue";:A$=INKEY$
WAIT FOR CHR(13)
WAIT @128,
WAIT @[15,4],"Hit <enter> to continue";FOR CHR$(13)
WAIT @[3,14],>MS$;FOR CHR$(13)
```

In the second example, A$ will contain the character hit on exit from the WAIT routine.

----------<ND86>----------

WORD POKE, WORD PEEK

```
WPOKE &H 40B1, 32000
A=WPEEK(&H4026)
```

WPOKE allows two bytes of memory to be changed with the one command. The WPEEK command complements the WPOKE command and allows the return of the contents (integer) of two bytes starting at the specified address. The format of the WPOKE statement is:-

WPOKE expl,exp2

where expl points at the first memory location to be poked, while exp2 is the data to be stored at those locations. WPOKE and WPEEK use the &% routine to evaluate the parameters, thus modulating them into INTEGER range.

These commands are mainly used when finding, or changing, vectors used by BASIC or DOS. See SETKEY (CH 12-7 lines 65290 - 65515) for a meaningful BASIC program example.

----------<ND86>----------

XOR -- See MOD (CH 9-11)

----------<ND86>----------

CHAPTER TEN

ADDED $TRING FUNCTIONS

The uses of MID$, LSET/RSET, and INSTR (which are normally available in Disk Basic) are explained here as well as Custom Basic's added string functions to help budding programmers.

BIT TESTING	INSTR(! allows programmer to set/reset/test bits contained in a string.
#DO	Executes a string as if it were a direct keyboard command.
FIELD@	Allows any area of memory to be treated like a string.
INSTR	Finds the occurrence of one string in another string.
INPUT TO USING	Allows fast Data entry routines (CH 11).
$LET	For more efficient use of string space.
LINELET	Allows any character to be stored in a string.
LOGICAL STRING	FUNCTIONS (AND$, OR$, XOR$)
LSET, RSET	Replaces string1 with string2 and blanks remaining portion of string1.
LSTRIP/RSTRIP	strip nominated characters from string ends.
MID$	Modifies string1 with string2.
ROT$	Rotate characters in strings.
SWP$	Swap characters in strings.
UPCASE$	Force all characters in a string to upper case.

NOTE: The Normal DOS routines of LSET/RSET and MID$ automatically move a string out of the program area. There are times when this is inconvenient. Therefore if the string variables are suffixed with a '*' they will not be moved. This allows you to edit a BASIC program using these commands. A prime use for this is if you use DATA statements as your means of data storage. INPUT TO also has this facility.

-----------<ND86>----------

BIT TESTING

```
A = INSTR(!var$, pos [,exp])
B = INSTR(!var$ AND exp$,exp)
INSTR(!var$,pos) = exp
```

This routine allows one to set up a string as a bit-map. Simply put, a bit-map allows the easy storage of information. This bit-map is stored in a string variable, maximum length of 255 characters, therefore the maximum number of bits is 255 * 8 or 2040 bits. The numbering convention is from left to right; bit 1 is the first bit of the string, bit 9 is the first bit of the second character of the string etc. Please note that this convention is totally different from the standard BASIC convention, therefore while the standard BASIC commands for setting of bits could be used, they are best avoided in order to prevent confusion.

Let's look at the examples:

Example 1:

```
A = INSTR(! A$, 12)
```

This example tests bit 12 of A$. If the bit is set, A will equal -1; if reset, A = 0.

Example 2:

```
A = INSTR(! A$,12,1)
```

This example will search A$ to find the first bit set starting with bit 12. If we change the 1 to a 0, it will find the first bit reset starting the search from bit 12. Note that an expression is allowed. If the expression is true then the string is searched for a set bit; if false a reset bit is searched for. On completion, A = 0 if no bit is found or A = the bit position of the first bit set or reset. E.g.: If bit 13 is the first set bit then A = 13.

Example 3:

```
B = INSTR(! A$ AND B$,1)
```

This example allows one to test several bits at once. It is easiest to set up a second string variable to hold the pattern you are searching for. If all the bits set in B$ are also set in A$, then B will equal -1; otherwise B = 0. If we substitute 0 for 1, then the test is made for a match between the reset bits of the strings.

Example 4:

```
INSTR(! A$, 12) = 1
```

This example will cause bit 12 of A$ to be set. If 0 is substituted for 1, then bit 12 will be reset.

NOTE: All the examples that have used 0 and 1 to indicate a set or reset bit actually use the truth or falseness of the expression. If the expression is true, the bit is set or a set bit is looked for. If the expression is false then the bit is reset or a reset bit is searched for. AND$,OR$ and XOR$ are also useful functions for bit-map manipulation. See CH 10-6.

----------<ND86>----------

STRING EXECUTION

#DO String expression

The #DO command allows BASIC to treat a string expression as though it were a direct command entered from the keyboard.

When used within a progam, the #DO statement must end in either a GOTO, RETURN or another #DO statement, otherwise program execution will cease after the end of the string being executed is reached. This is due to the manner in which the routine is handled by the interpreter.

This causes very little hardship, as the #DO statement line simply becomes a line to which GOSUB's are made. For example, assume that you want to be able to process a user inputted formula within a program, we can use the #DO in the following manner:-

```
 10    INPUT "Enter the Formula ";H$
 20    GOSUB 100
 30    PRINT H$;" when evaluated returns ";D
 40    GOTO 10
100    #DO "D = " + H$ + ":RETURN"
```

The only limitation on the instructions used in the #DO command is that since the #DO execution is considered to be a direct command statement, BASIC statements that would normally give an ILLEGAL DIRECT error may not be used (e.g. INPUT).

#DO complements the DEFFN statement, since the DEFFN statement may only be a mathematical expression whereas the #DO can contain BASIC commands such as IF...THEN...ELSE as well as any EXTENDED BASIC command including DEFFN and #DO.

Quite complex command sequences can therefore be set up by including multiple #DO statements inside IF...THEN...ELSE structures.

NOTE: A #DO statement terminates execution of any previous #DO statement. Since the routine must tokenise the string each time, program execution will be slow as the tokeniser must be loaded from disk.

----------<ND86>----------

=============== NEWDOS86 (C) 1986 W.S.& D.S. SANDS ================

FIELDING STRINGS

FIELD@ &H1929, A$ LEN 5

This command allows you to point or, more correctly, field a string variable at any memory location. The memory area may be ROM (for reading data) or RAM (including the video display). The length of the string is still limited to the standard BASIC length of 255 characters. The command format is:-

FIELD@ address, var$ LEN exp [, var2$ LEN exp2...]

Once a string variable has been fielded at a portion of memory, that memory may be manipulated by using any of the standard BASIC string handling commands.

In the example above, the contents of A$ is READY (stored in ROM at &H1929). While the ROM contents can't be modified, areas of ROM do contain much data that you may desire to use.

The string will remain pointing to the memory area specified until another assignment of the variable occurs. However, BASIC's string handling routines will normally move any string located within the program into string space before altering it, preventing corruption of the BASIC program.

The LSET, RSET and MID$ commands are normally used to change or modify strings created with FIELD@. If you wish to modify strings within the program itself, don't forget to add the '*' suffix to the variable name. This will inhibit any movement of the string.

One problem with the screen display routine is that if you try to print anything to the last video location the screen display always scrolls. The FIELD@ statement can alleviate the problem, as the following example will show.

```
FIELD@ 16320, BL$ LEN 64      'BL$ points at last line of screen
RSET BL$ = "123"              'blank the line and right justify the
                               '123'
MID$(BL$,1) = "ABC"           'print 'ABC' at bottom left of the
                               screen
```

This command opens up many interesting ways to manipulate memory. The various uses to which it can be put is limited only by your imagination.

----------<ND86>----------

INSTR FUNCTION

A = INSTR(2,A$,"8")

This allows you to search a string for a specified sub-string and the format for this new command is:-

var = INSTR([exp,] expl$, exp2$)

exp = the position in the string, expl$, at which to begin the search. If not specified, exp defaults to 1.

expl$ = the string being searched

exp2$ = the string we are looking for

The result of this function specifies the position in the string at which the first match is found. If no match occurs a value of zero is returned.

Another feature which increases the versatility of my routine is the ability to specify a wild character, by the use of the "*". This allows partial matches to be found.

For example:-

```
1Ø      A$ = "ABCDEFBCEFBZD"
2Ø      B$ = "BC"
3Ø      PRINT INSTR(A$,B$)           ...without exp
4Ø      PRINT INSTR(3,A$,B$)         ...with exp
5Ø      PRINT INSTR(A$,"BCD")        ...without exp
6Ø      PRINT INSTR(8,A$,"F**D")     ...with exp

RUN

2       ................................result of line 3Ø
7       ................................result of line 4Ø
2       ................................result of line 5Ø
1Ø      ................................result of line 6Ø
```

NOTE:-

1) Once a substring is found, the search is discontinued -- only one result will be returned each time the routine is accessed.

2) If the searched for string (exp2$) is a null string, the result will always equal the value of exp.

For BIT TESTING with INSTR(! see CH 1Ø-2.

----------<ND86>----------

$LET var$ = string expression

Will assign the string expression to var$. If the string expression will fit in the current var$, then no new strings are created. If the string expression is longer than var$, then var$ is moved so that all temporary strings are lost.

It is recommended that you use $LET whenever you are assigning a string to itself. E.g.:

```
$LET A$=UPCASE$(A$)
```

----------<ND86>----------

LINELET A$ = [Rem] line

This allows ANY information to be stored in a string variable. If the optional REM is incorporated, the string will start with the first character following the REM and the line will remain in ASCII form. If the REM is not used, the line will be tokenised, allowing graphic characters to be stored in the string. The string is considered to be all characters following the '=' or the 'REM' up to the end of the line (but not including the carriage return). This allows multiple quotation marks to be entered into the string. E.g.:

```
10    LINE LET A$ = THIS IS A TEST OF THE "LINE LET"
20    PRINT A$

THIS IS A TEST OF THE "LINE LET"
```

----------<ND86>----------

LOGICAL STRING FUNCTIONS

It is now possible to AND, OR and XOR string expressions. Due to limitations imposed by the ROM, the syntax unfortunately is different from the standard ROM LOGIC functions. The syntax is:

```
AND$(exp$1,exp$2)
OR$ (exp$1,exp$2)
XOR$(exp$1,exp$2)
```

The output of the function is a string whose length is equal to the longer of the two expressions. So if you are performing a masking function, use LSET, RSET or MID$ to maintain the original string length. Neither of the two input strings are affected in any way.

```
10 A$="@@@"
20 PRINT OR$(A$,CHR$(1))
A@@
```

----------<ND86>----------

LSTRIP & RSTRIP

var$=LSTRIP(exp$[,exp]):var$=RSTRIP(exp$[,exp])

These two functions allow you to strip characters from the left or right of strings. The optional exp allows you to specify the character/s you wish to lose and if not specified defaults to 32 (a space).

```
1Ø  A$="***.A.test"
2Ø  B$=".of.the......"
3Ø  PRINT LSTRIP(A$,42)+RSTRIP(B$,ASC("."))+".function"
4Ø  PRINT A$,B$
RUN

.A.test.of.the.function
***.A.test    .of.the......
```

The asterisks were removed from the left side of A$ and the periods from the right side of B$. If you wish, you can strip both ends of the string by combining the LSTRIP and RSTRIP commands.

Neither A$ nor B$ were affected by the command. If you wish to store the stripped string back to itself, use $LET to conserve string space.

```
A$=".....A test....."
$LET A$=LSTRIP(RSTRIP(A$,ASC(".")),ASC("."))
```

----------<ND86>----------

=============== NEWDOS86 (C) 1986 W.S.& D.S. SANDS ================

LSET,RSET

```
LSET A$ = B$
RSET A$ ="123"
```

LSET 'left justifies' data in a string variable, while RSET 'right justifies' the data. If the data item is too long to fit in the string variable, it is always truncated on the right. That is, the excess characters on the right are ignored. The format of both commands are:-

```
LSET  var$[*] = exp$
RSET  var$[*] = exp$
```

where var$ is any string variable and exp$ is any allowable string expression. The optional star suffix allows strings within the program area to be altered. The easiest way to explain what LSET/RSET do is to give a few simple examples.

```
 1Ø    A$="ABCDE"
 2Ø    GOSUB 2ØØ
 3Ø    LSET A$ ="123"
 4Ø    GOSUB 2ØØ
 5Ø    RSET A$ = "123"
 6Ø    GOSUB 2ØØ
 7Ø    LSET A$ ="1234567"
 8Ø    GOSUB 2ØØ
 9Ø    RSET A$ ="1234567"
1ØØ    GOSUB 2ØØ
11Ø    END
2ØØ    PRINT "!";A$;"!"
21Ø    RETURN

RUN

!ABCDE!
!123  !
!  123!
!12345!
!12345!
```

It is apparent that LSET and RSET behave identically when the string expression is longer than the string variable. As with the expanded MID$ command, no new strings are created -- thus reducing the number of string space compressions and thereby speeding up program execution when large numbers of strings are being used.

-----------<ND86>-----------

MID$ FUNCTION

MID$(A$,1,4) = "AB"

CUSTOM BASIC gives improved string handling capabilities by being able to use MID$ on the left-hand side of an assignment statement, as well as on the right-hand side. This allows the replacement of portions of one string with another string.

The format of the statement is:-

MID$(var$ [*],exp1 [,exp2]) = exp$

where var$ = the string to be changed.

exp1 = the position in var$ to start the change

exp2 = the number of characters to change. [optional]

NOTE: The number of characters to change always takes on the smallest value out of these parameters:-

1) The length remaining in var$
2) The value specified by exp2
3) The length of exp$

For example:-

```
1Ø      A$ = "FRED IS A GOOD BOY"
2Ø      PRINT A$
3Ø      MID$(A$,1) = "BILL"
4Ø      PRINT A$
5Ø      MID$(A$,1,4) ="GREGORY"
6Ø      PRINT A$
7Ø      MID$(A$,1,4) = "JOE"
```

RUN

```
FRED IS A GOOD BOY          ....line 2Ø
BILL IS A GOOD BOY          ....line 4Ø
GREG IS A GOOD BOY          ....line 6Ø
JOEG IS A GOOD BOY          ....line 7Ø
```

Note that only three characters were replaced in the last example. Also since the routine actually replaces the characters in var$ with exp$, BASIC's string reorganisation routine is not accessed. Therefore if this MID$ function is used wherever possible, the number of those interminable delays will be reduced.

----------<ND86>----------

ROTATING CHARACTERS IN STRINGS

var$=ROT$var$ [,exp]

This command will cause a string to be rotated exp number of bytes. A positive exp, will cause the string to be rotated from right to left, while a negative value will rotate it left to right. Exp defaults to a value of 1.

```
10 A$ = "12345678"
20 PRINT A$
30 A$ = ROT$(A$)
40 PRINT A$
50 A$ = ROT$(A$,-3)
60 PRINT A$
Run
12345678
23456781
78123456
```

Three main areas of applications arise for this routine, one is in bit manipulation, two is manipulation of graphics characters contained in strings, and the other is in the simplicity of scrolling of messages across the screen (in conjunction with FIELD @).

----------<ND86>----------

STRING SPACE SAVER

Automatically causes all temporary strings to be removed whenever the '*' suffix is used in the LSET/RSET and MID$ commands. Also all temporary strings generated by PRINT (and PRINT USING) will be removed.

----------<ND86>----------

SWAPPING CHARACTERS IN STRINGS

$SWP var$, pos1,pos2,number of bytes to swap

Swaps the specified number of bytes from position1 to position2. E.g.:

```
A$=RIGHT$(A$,4)+MID$(A$,5,LEN(A$)-8)+LEFT$(A$,4)
```

can be replaced with

```
$SWP A$,1,LEN(A$)-4,4
```

will swap the first four bytes with the last 4 bytes. If the bytes to be swapped overlap, a ?FC error is returned. E.g.:

```
A$ = "123456":$SWP A$,1,4,7
```

is invalid. The main advantage of $SWP over the MID$ method is that no new strings are generated and therefore 'garbage collections' are reduced.

----------<ND86>----------

UPPER CASE CONVERSION

UPCASE$(exp$)

Converts all characters in the string expression to their uppercase equivalents. For the purposes of the conversion, bit 7 of the character is ignored. E.g.:

```
UPCASE$( CHR$(97) )       returns CHR$(65)       ("a" = "A")
UPCASE$( CHR$(97+128) )   returns CHR$(65+128)
```

This allows simplified INKEY$ testing, particularly if you are using the Control Key Driver to provide function keys. As with the logical string functions, the original string expression isn't affected by the conversion.

```
    A$ = "aBcd"
    B$ = UPCASE$(A$)
    PRINT A$,B$

aBcd   ABCD
```

----------<ND86>----------

CHAPTER ELEVEN

FORMATTED INPUT ROUTINE

INPUT TO ...USING...

CUSTOM BASIC provides a very powerful and much improved INPUT routine. It is easy to use! You may end up having some quite long strings in your program. Once you have made up these strings you learn to forget the 'internals' of the string and think only in terms of what the string does. Please read on. We will start off simply and work up to some quite complex input routines.

The standard BASIC routines, INPUT and LINEINPUT, while in many cases quite satisfactory, have many undesirable features, such as the <clear> key erasing the screen, the <shift right arrow> initiating 32 character mode and the <enter> key generating a linefeed on the display. Normally the only way to overcome these limitations is to use a BASIC routine built around the INKEY$ statement. The problem with this technique is that it takes a great deal of code to implement and keyboard response becomes exceedingly sluggish.

This formatted INPUT routine overcomes all of the nasties of BASIC's routines, with no noticeable degradation of keyboard response time. The command format of the routine is:-

```
INPUT TO [@location,] ["prompt string";] var1$[*] [;] [USING var2$]
```

The first string variable, var1$, we will call the Target string (T$), since what the operator types will be stored directly into this string. The length of this string determines the number of characters that can be entered from the keyboard. A FUNCTION CALL ERROR is returned if T$ has zero length.

Don't let the great number of options scare you. They are provided to simplify your BASIC programs. Let's RUN a bare bones example of INPUT TO which uses none of the optional features:-

```
10    T$ = "1234"
20    INPUT TO T$
30    PRINT T$, LEN(T$)
```

Notice that you can't type in more than 4 characters. The <break>, <down arrow>, <right arrow> and <clear> keys have NO effect and linefeeds are not generated when you hit the <enter> key. This means that you won't have your screen layouts destroyed each time data is INPUT, and since the <clear> and <down arrow> keys don't work, the operator can't destroy the screen display either. Note that it is now possible to input leading spaces into strings without having to use double quotes (") and double quotes can be included in the string.

Let's examine some of the options. You can now specify a screen address at which the INPUT is to occur, using the same syntax as the PRINT@ statement. This now has two forms -- the old PRINT@454, and the new PRINT@[7,6]. Substitute this line 2Ø in the example above:-

```
2Ø    INPUT TO @[7,6], T$
```

Notice that the cursor always first appears at the screen position 454. The optional prompt string is identical to that used in the standard BASIC INPUT statement:-

```
2Ø   INPUT TO "ENTER THE DATE PLEASE";T$
```

Obviously you can combine both options thus:-

```
2Ø   INPUT TO @454,"ENTER THE DATE PLEASE"; T$
```

If you suffix T$ with a '*', then you can actually modify strings located in the BASIC program. Try the following lines:-

```
2Ø   INPUT TO @454,T$*
4Ø   LIST
```

notice that the string in line 1Ø of the BASIC program now becomes whatever was typed in from the keyboard.

The final option concerned with T$ is the semi-colon. If this is used then the contents of T$ will be printed at the current cursor position. If the optional prompt string is used, T$ will be printed immediately after the prompt string. Thus:-

```
2Ø   INPUT TO @454,"ENTER THE DATE ";T$*;
```

will cause "ENTER THE DATE " to be printed at 454, followed by the contents of T$. If you don't want T$ to be printed immediately after the prompt string, make the last character of the prompt string a space.

We shall now discuss the USING C$ option. In the above examples the machine code routine supplied a default control string (C$). In many cases this default string is quite satisfactory. However, to allow you even more control over what is INPUT, you can specify your own C$.

C$ allows you to specify where in T$ the first character typed is stored. It also lets you determine what are acceptable input characters. By setting up the proper C$ you can cause the routine to accept only the numerals Ø-9 and ignore all other characters. You can also indicate what characters are to end the INPUT (these characters are called 'exit characters'). This allows you to allocate specific keys for specific tasks (or functions).

For example: Assume that you are writing a mailing list program. Creating your own control mechanism will allow you to assign the function of scrolling through your records to the <up> and <down> arrows. You could also use the <^S> key to save the current record and <^F> to find a particular record.

Since the routine overwrites an already existing string, the question arises: If an exit character is entered before the end of the string is reached -- what do we do with the remaining portion of the string? Several alternatives exist:

1) The string could be shortened so that only those characters up to the exit character are considered to be part of the string. This is how the standard INPUT routine in BASIC works.

2) All characters from the exit character onwards could be blanked out or replaced with another character.

3) The routine could simply exit and not affect the string at all.

To make this routine as versatile as possible, any one of the above options is available for each exit character. Thus the <up> and <down> arrow keys may use option 3 while the <enter> key could use option 1.

To give more versatility the <backspace> key can give you either destructive or non-destructive backspace capability. If a non-destructive backspace is chosen, the <right arrow> key moves the cursor forward through the text. If destructive backspace is chosen, the <right arrow> is ignored. You can even select the character that does the erasing. Use a CHR$(32) if you wish to emulate the standard BASIC backspace action.

As well, you can select whether to exit at end of field or not. You can exit from the left of the field, from the right of the field or from both left and right sides.

The format for C$ is presented on page CH 11-8. While it may look somewhat complex, after you have worked through the following examples I feel sure you won't have problems designing your own C$.

The examples are heavily commented and for the sake of understanding and readability we have not put all the remarks at the end of the statement. To avoid confusion as to what is program and what is comment, the program is in upper case while the comments are in lower case. Note also that we won't be using all the available options in all of the examples. We leave it to you to experiment.

It is desired to have a maximum of four characters entered into the string variable T$, the only character you desire to terminate data entry is the <enter> key. Destructive backspace is desired and only Hexadecimal characters are acceptable as input. The first character entered is to be the first character of the input string.

A BASIC program to implement these requirements would be as follows:-

```
10  CLS:CLEAR 2ØØ:T$=STRING$(4,95)
                          Only accept 4 characters

2Ø  C$=CHR$(1)            start inputting to the first character
       +CHR$(Ø)           provide storage space for the last character
                          typed
       +CHR$(1)           only one exit character
       +CHR$(13)          exit on <enter>
       +CHR$(28)          modify the string length on exit
       +CHR$(95)          destructive Backspace
       +CHR$(29)          don't exit at end of field
       +CHR$(3)           three sets of limit characters
       +"AFØ9af"          they are Ø-9, A-F and a-f
3Ø  INPUT TO @[7,6],"ENTER HEX NUMBER ";T$; USING C$
                          Print prompt string at 454, then print 4
                          underline characters and accept the input

4Ø  PRINT:PRINT T$        display the input

5Ø  END
```

A slightly more complicated example -- it is desired to INPUT a string of a maximum of fifteen characters. The exit characters are to be <break>, <enter>, <up arrow> and <down arrow>. <break> and <up arrow> are to exit, while <enter> is to blank the string and <down arrow> should truncate the string. Non-destructive backspace is required. Alphabetic characters together with the space are acceptable. The <break> key terminates the program. Additionally we require the <^F> key to activate a Find function and the <^S> to save the current record.

```
1Ø CLS:CLEAR 2ØØØ
11 SETKEY 1,CHR$(198) GOSUB 12ØØ          the <^F>ind routines
12 SETKEY 2,CHR$(211) GOSUB 13ØØ          and <^S>ave routine

2Ø C1$=CHR$(1)          <break> as exit
     +CHR$(Ø)           do nothing on exit
     +CHR$(13)          <enter> key
     +CHR$(32)          blank the string on exit
     +CHR$(1Ø)          <down arrow> key
     +CHR$(28)          truncate the string on exit
     +CHR$(91)          <up arrow> key
     +CHR$(Ø)           take no action on exit
```

by making our exit characters a sub-string, processing is simplified further on in the program.

```
30  C$=STRING$(2,1)    start with first character in field
       +CHR$(LEN(C1$)/2)   set the number of exit characters
       +C1$            add the exit characters
       +CHR$(24)       non-destructive backspace
       +CHR$(29)       don't exit at end of field
       +CHR$(255)      now to flag the range of
       +CHR$(128)       exit characters.  Exit on all
       +CHR$(255)       characters between 128 and 255
       +CHR$(0)         without affecting T$
       +CHR$(3)        three sets of limits
       +"af  AZ"       and then the limit characters
50  WHILE 0            set the loop condition
55  T$=STRING$(15,35)  sets the length and displays "#"'s

60  INPUT TO @454, T$; USING C$

70  ON (INSTR(C1$,MID$(C$,2,1))+1)/2 GOSUB 100,200,300,400
```

As the exit character is stored in byte 2 of C$ the INSTR function allows a simple means of processing the various exit characters. The SETKEY routine will look after any specialised functions we might want to program into the routine.

```
80  PRINT@ 110, T$     print the inputted string

90  WEND               continue until <break> is hit.
                          Try swapping lines 50  and 55.

100 END

200 PRINT@ 20,"<enter> PROCESSING OCCURS HERE.....";:RETURN

300 PRINT@ 20,"<down arrow> PROCESSING OCCURS HERE";:RETURN

400 PRINT@ 20,"<up arrow> PROCESSING OCCURS HERE..";:RETURN

1200 PRINT@20,"The Find function                   ";:ENDKEY

1300 PRINT@20,"The Save function                   ";:ENDKEY
```

As you can see, the INPUT TO routine isn't all that hard to use -- particularly when you consider all the control it gives you over data input. If you do experience difficulties, check the length of T$ and then check that your control string is correct. An incorrect C$ is the normal cause of most errors.

Before we leave the FORMATTED INPUT ROUTINE we should discuss the limitations of it. Since it uses the CHR$(24) and CHR$(25) functions to tab and backspace through the line it can only work successfully on a single video line. If you wish to use the routine over several lines, use an array mechanism to cope with the multi-line concept.

What follows are a few tips that I have picked up in using this CUSTOM BASIC routine:-

Firstly: Store all your control strings in arrays. I have created for myself a 'library' of control strings. I merge these with all programs I am writing. Doing this simplifies the creation of input routines. Once the program has been debugged, I save all the control strings in a small MU file, and then delete the original definitions. The control strings are read in by the program during initialisation. This frees up quite a bit of program space.

Secondly: To handle input of dates and numbers where separators are used (e.g. 1Ø/11/88 and 123.45), I normally split the field into several fields. This allows the routine to 'step over' the separator.

Thirdly: When you are dealing with several screens having many fields, the use of arrays to hold the INPUT USING data is recommended.

What follows is the data entry portion of a program I have written that uses INPUT..TO. This program, SCHEDULE/BAS and STUDENT/DAT, are included on the supplied disk. Run it sometime and see what it does. The requirement was to make up a weekly schedule for an organisation that runs a comprehensive course in public speaking. The first two talks are given by 'instructors' to the entire group. After this the group splits into three sub-groups or schools.

Individuals progress through the school up to the point of being instructors themselves. SCHEDULE/BAS is used to enter data as to what talks the various students are eligible to perform. I haven't included the entire listing here due to space considerations, but please list out and peruse the program at some time. However I will discuss the control string, since it incorporates a trick worth remembering.

Briefly, PN%(,) holds the PRINT@ position and also the index of the USING string to use for each field. NN$( ) is the screen variable, KD$( ) is the USING string and GG is the field counter. There is a list of variables and their uses at the end of the program.

Since we want the backspace key to allow us to move across field boundaries, we include its code in CK$(Ø). We do an INSTR search of CK$(Ø) to determine which exit key was pressed and then do a GOSUB to the appropriate routine. But we can't include the backspace code into the USING string. If we did we couldn't backspace within the field.

```
61Ø28 'CK$(Ø)    1   Backspace
                 2   Clear
                 3   Up arrow
                 4   Shift left
                 5   Shift right
                 6   down arrow
                 7   enter
                 8   break
                 9   shift up arrow
                1Ø   shift down arrow
61Ø3Ø CK$(Ø)=CHR$(8)+CHR$(Ø)+CL$+CHR$(Ø)+ES$+CHR$(Ø)+CHR$(24)+
CHR$(Ø)+CHR$(25)+CHR$(Ø)+CHR$(1Ø)+CHR$(Ø)+CR$+CHR$(32)+CHR$(1)+
CHR$(Ø)+CHR$(27)+CHR$(Ø)+CHR$(26)+CHR$(Ø)
61Ø4Ø CK$(1)=CHR$(24)+CHR$(Ø)'backspace function
61Ø5Ø CK$(2)="  azAZ''" 'Name entry
61Ø6Ø CK$(3)="Ø9  "  'Numeric entry
```

So when we come to create the USING string, we skip the backspace code by using the MID$() function.

```
61Ø7Ø KD$(Ø)=STRING$(2,1)+CHR$(LEN(CK$(Ø))/2-1)+MID$(CK$(Ø),3)+
CK$(1)+CHR$(255)+CHR$(128)+CHR$(255)+CHR$(Ø)+CHR$(LEN(CK$(2))/2)+CK$
2)
```

As you can see the actual INPUT code is very similar to the example above.

```
1ØØ3Ø DO:INPUT@PN%(GG,Ø),TO NN$(GG)*; USING KD$(PN%(GG,1))
1ØØ4Ø $LETA$=MID$(KD$(PN%(GG,1)),2,1)
1ØØ5Ø PRINT@PN%(GG,Ø),NN$(GG);
1ØØ6Ø A=(INSTR(CK$(Ø),A$)+1)/2
1ØØ7Ø IF (A<6 AND A>Ø) THEN ON A GOSUB 1Ø6ØØ,1Ø5ØØ,1Ø6ØØ,1Ø7ØØ,
            1Ø75Ø ELSE GOSUB 1Ø65Ø
1ØØ8Ø UNTIL GG>5 OR A>8
1ØØ9Ø RETURN
```

The control string can be sub-divided into the following manner:-

```
A  B  C  Dd  ..  .. ..   E   F   Gghi .... ....   J  Kk .. ..  ..

A.... This byte always contains the position in T$ where the next
      character received will be stored.
B.... This byte holds the last character typed. This byte can
      also be retrieved by the INKEY$ function and allows you to
      decide if further action needs to be taken.
C.... This byte holds the number of exit character pairs.
Dd... The exit character pairs.
   D  The exit character, e.g. CHR$(13) exit on <enter>
   d  What to do to T$:-
           CHR$(Ø)   don't alter T$
           CHR$(28)  reduce the length of T$
                     else overwrite the rest of the T$ with CHR$(x)
E.... This controls the action of the <left arrow> key.
           CHR$(=>32)  destructive backspace. The CHR$ specified is
                       used to overwrite the previous character.
           CHR$(<32)   non-destructive backspace and the <right
                       arrow> is activated.
F.... Exit at end of field function.
           CHR$(Ø)     exit field at either end
           CHR$(1)     exit allowed only at the left of field
           CHR$(2)     exit allowed only at the right of field
           CHR$(29)    don't exit at end of field. If exit is
                       allowed at end of field then the routine
                       exits once the field is filled, otherwise the
                       routine waits for an exit character.
Gghi  The optional range exit fields:-
   G       Always a CHR$(255)
   g       Lower bound of the exit range
   h       Upper bound of the exit range
   i       The action to take on exit. Same as d above.
J.... This byte stores the number of limit character pairs
           specified.
Kk... Then follow the limit character pairs. In ascending ASCII
           order.
   K       The lower acceptable character
   k       The upper acceptable character
```

The codes Ø,1,2,28 and 29 were chosen as control codes as it is highly unlikely that these values will ever be used as exit characters. This allows the INSTR function to be used, as in line 70 of the example program CH 11-5.

----------<ND86>----------

CHAPTER TWELVE

NEW CONTROL STRUCTURES FOR STRUCTURED PROGRAMMING

WHILE:WEND

WHILE exp:code to do....:WEND

This routine will test whether the expression is true or false. If the expression is true then the code following the WHILEexp: will be executed, otherwise all program up to the matching WEND (While END) it encounters will be skipped.

Each time the WEND is encountered, the expression is once more evaluated. If it is still true the code is once more executed, otherwise the program continues from the WEND. A simple example:

```
120 WHILE A<10+
130 PRINT A+
140 INC A+
150 WEND+
160 PRINT"END!"+
```

NOTE: The expression can be a variable, e.g. WHILE A :.....:WEND; while A is non-zero the expression is true.

A slightly more complicated example. This is a simple SHELL SORT program. The variable TR contains the subscript value of the last element in the array. SE$ is the match we are searching for and CP$() holds the list of names.

```
2999 'Search routine+
3000 CLEAR 10000:DEFINT A-Z:DIM CP$(100):#RESTORE 3200:TR=0+
3010 DO:READ CP$(TR):INC TR:UNTIL CP$(TR-1)="End"+
3020 INPUT "Enter the search string ";SE$:$LET SE$=UPCASE$(SE$)+
3030 I=0:K=TR-1:KI=K  'Set the initial values+
3040 WHILE K>I AND CP$(KI)<>SE$+
3050 KI=(K-I)/2+I+
3060 IF SE$>CP$(KI) THEN K=KI-1 ELSE I=KI+1+
3070 WEND+
3080 IF CP$(KI) = SE$ THEN PRINT "A Match has been found"+
     ELSE PRINT "That Name is not in the list"+
3200 DATA "ANDREW","BOB","CHARLES","DON","EDWARD","FRED"+
3210 DATA "MICHAEL","PETER","STEPHEN","ZORRO","End"+
```

-----------<ND86>----------

DO:UNTIL

DO:..........:UNTIL exp

This command is similar to the WHILE:...WEND; the only difference is that the test is done at the end of the code and not at the start. This means that the code between DO .... UNTIL will always be executed at least once.

```
  10   DO+
  20   READ A$+
  30   H=ASC(A$)-48:IF H>9 THEN H=H-7+
  40   L=ASC(RIGHT$(A$,1))-48:IF L>9 THEN L=L-7+
  50   PRINT H,L,H*16+L+
  60   UNTIL A$="END"+
  70   PRINT"END OF ROUTINE"+
  80   more code follows
2000   DATA FF,12,34,15,28,....END+
3000   END+
```

WHILE....WEND and DO...UNTIL can be nested to any number of levels. The only limitation is due to the memory used, since each level creates an entry on the stack which consumes 7 bytes. A simple one liner to check out what values are returned by various keys:-

```
DO:DO:A$=INKEY$:UNTIL A$>"":PRINT ASC(A$);:UNTIL A$=CHR$(0)
```

A handy wait loop testing for <ENTER>:

```
DO:UNTIL INKEY$=CHR$(13)    [DO nothing UNTIL <ENTER> is
                             pressed]
```

Any key could be used by substituting its ASCII value for 13. A great advantage of this routine over the usual GOTO type is that it can be in the middle of a compound statement line -- also it doesn't get in the way of TRON in this circumstance.

----------<ND86>----------

CASE

```
CASE  expl
   ;exp2,exp3:code to do
   ;=>exp4,<exp5:code to do
   ;ELSE code to do
ENDCASE
```

Nearly all programs require the selection of one of a set of actions according to the value of an expression. BASIC already provides ON...GOSUB, ON...GOTO and IF...THEN...ELSE. But at times these mechanisms just can't cope with the either the number of choices or the complexity of the choices. Most programmers have spent much time trying to see why heavily nested IF...THEN...ELSE program lines are not working as they should.

The CASE structure is designed to simplify this choice mechanism that is so often necessary in programming. To aid in the discussion let's examine how the CASE structure works. The concept is simple and is closely related to the IF...THEN... ELSE statement.

```
IF a statement is true
    THEN execute the code for true
    ELSE execute the code for false
```

Thus we see that the IF statement evaluates and tests an expression. This can have two possible outcomes, true or false. Each outcome has associated with it a block of code, or LIMB. The true LIMB is indicated by THEN while the false LIMB is flagged by ELSE. Notice that one and only one of the two LIMBS is executed, never both.

Let's examine CASE. The CASE statement evaluates and stores an expression. But the test is made whenever a new LIMB is encountered and, as can be seen, the type of test to make is stored at the beginning of the LIMB. When the test succeeds, the LIMB is executed and program execution resumes after the matching ENDCASE statement. Otherwise the next LIMB is searched for.

The semi-colon as the first character in a statement indicates that a new LIMB has been encountered and if not followed by < = or > also implies that the test is for equality.

CASE is easily understood from the following simple example:

```
10  B=16:CASE A%+
20  PRINT "A% is ";+
30   ;1,2:  PRINT "1"+
40    PRINT" or A% is 2"+
50   ;<1:   PRINT"less than 1"+
60   ;>B:   PRINT "Greater than";B+
70   ;<=5:  PRINT "3, 4 or 5"+
80   ;ELSE  PRINT "between 6 and";B+
90  ENDCASE+
```

It is obvious that such a simple task could be done with the IF statement if we re-arranged the program. The major advantage of the CASE structure is that it allows the LIMB to span several BASIC program lines. Naturally, the entire example could have been written as single line if we so desired.

The CASE statement can be nested to any number of levels with the proviso that if string expressions are always used as expl, then STRING TOO COMPLEX ERROR may be encountered after about the eighth active CASE statement. The error END OF PROGRAM ENCOUNTERED will be given if there is no matching ENDCASE statement and if CASE isn't active and a semi-colon or ENDCASE statement is encountered, the message CASE STATEMENT NOT ACTIVE will be given.

Let's now examine the major limitation of CASE -- and how to overcome it. CASE cannot directly test for a range of values since the test implies the OR mechanism. Line 2Ø could be read as IF (A%=1 OR A%=2) THEN ... whereas to test for a range of values we need to be able to imply IF (A% > 1 AND A% < 4) THEN...

This was overcome in the example by arranging the tests in a particular order so that the test in Line 7Ø would only apply to 3,4 or 5 since lines 3Ø through 5Ø had already coped with the other values.

An alternative method is to change the argument of the CASE statement. In the example we made A% the argument and all tests involved A%, but if we make TRUE (-1) or FALSE (Ø) the argument, then we can test for anything. Thus:

```
1Ø  CASE  -1+
2Ø  ;( A% >1 AND A% < 4 ) : LIMB code 1+
3Ø  ;ELSE :LIMB code 2+
4Ø  ENDCASE+
```

----------<ND86>----------

PROGRAM SPEED COMPARISONS

A few comments on the relative speed efficiency of the WHILE, DO and CASE statements compared with each other and the FOR...NEXT loop and IF...GOTO statement:

The FOR...NEXT loop would probably be the fastest, since only one variable has to be found and the STEPping of the variable occurs within the routine. The DO...UNTIL command must come a very close second, since it can also reference a single variable, but the STEPping of the variable must be done by the BASIC programmer. Both of these are fast since the code is always executed once and therefore the interpreter knows where the beginning of the loop is.

WHILE...WEND versus the GOTO. For the GOTO the speed of execution depends on the size of the BASIC program and the position in the program of the code block to be skipped. The WHILE...WEND's efficiency depends on the size of the code between the WHILE and the WEND. If the WHILE...WEND will most likely execute the code at least once, then the WHILE...WEND will always be faster than the GOTO since the start of the loop is known when the end of the loop has been reached.

CASE versus GOTO. This is a little different to the WHILE...WEND versus GOTO. Since only one LIMB will ever be executed, the other non-executed limbs must be skipped. This involves a character by character search for the various CASE keywords (; and ENDCASE). Therefore it is possible that the GOTO may always be faster than the CASE, particularly if the code involved is at the front of the program, since the GOTO would have very few lines to search and could "leap great blocks of code in a single bound". CASE (and also WHILE...WEND) could have been speeded up by demanding that the CASE keywords be the first statement on the program line, but this would limit the use of the CASE statement.

SEE also CH 8-2 and CH 8-4.

----------<ND86>----------

SETKEY

Setkey has two modes:

1) Mode 1 occurs while at BASIC ready and gives shift keyword entry and automatic case toggle. See Chapter 5-5.

2) Mode 2 occurs when a BASIC program is executing. The SETKEY command allows you to force your program to execute one of 8 subroutines whenever the associated key is pressed. The syntax is:

```
SETKEY exp1 [,exp$ [GOSUB exp2]]
ENDKEY
SETKEY = y/n
var = KEYVAL exp1
```

exp1 must be in the range of 1-8 and indicates the priority of the KEY. KEY 1 has the highest priority, while KEY 8 has the lowest. The key you wish to check for is specified by exp$, this value is forced to uppercase (in the same fashion as UPCASE$). Exp2 is the line number of the routine you wish to branch to. For example:

```
SETKEY 2,CHR$(194) GOSUB 1ØØØ
```

will cause your program to execute the code at line 1ØØØ whenever you hit <^B> or <^b>. The routine is terminated by the ENDKEY statement. This functions in a similar manner to the RETURN instruction.

SETKEY = y/n
allows you to enable/disable all the keys at once. If issued within a SETKEY routine, it remains in force until the ENDKEY instruction is encountered. At this time it reverts to the status it had before the routine was entered. More on this in a moment. RUN and CLEAR do an internal SETKEY=Y.

```
SETKEY 1
SETKEY 1,CHR$(Ø)
SETKEY 1,CHR$(x) GOSUB Ø
```
all disable the key specified, in this case KEY 1.

var = KEYVAL exp1
allows you to store in the specified variable, the current line number assigned to KEY exp1.

Before we examine some program examples, it behoves us to examine how the routine functions. At the end of every BASIC program statement (e.o.line or :), the BASIC interpreter scans the keyboard to test for <break> or <|@>. Also at this time if a key has been pressed, it is stored ready for a possible INKEY$ request.

SETKEY uses this point to check if a valid user specified key has been pressed. If so, it then does the following:

1) It stores the current priority.
2) It invalidates all keys of this or lower priority.
3) It does, in effect, a GOSUB to the line specified.

When the ENDKEY instruction is encountered, the routine exits and restores the previous priority.

Another point of interest. When the Keyboard Type Ahead is installed, BASIC normally does not scan the keyboard at e.o.statement. This allows you to enter multiple INKEY$ characters, which means that by the time BASIC has finished loading, you can be 'positioned' at the fourth menu. When SETKEY is active, the keyboard is scanned at the e.o.s. and this 'empties' the KTA. An alternative to SETKEY (for simple applications) is to use the byte at 4423H. PEEKing this can inform you if any of the <shifted arrow keys> have been pressed. See the technical section (CH 3-2) for further information.

What uses are there for the SETKEY command? It can be very handy in debugging a program. For this reason I normally try to retain KEYs 1,2 and 3 for debugging purposes. Another use is if your program is editing disk file data. Newdos8Ø/86 allows you to force a write of the current file sector to disk with a PUT,fan,&. By making this a SETKEY accessed routine, you can save your data to disk whenever you feel the need. Another use is in menu driven programs, where you desire to incorporate an escape mechanism to exit out of the menus.

Now for some examples. (1) Debugging:

```
1 SETKEY1,CHR$(193) GOSUB 65ØØØ
   :SETKEY2,CHR$(194) GOSUB 651ØØ
   :SETKEY3,CHR$(195) GOSUB 652ØØ:DIM CP%(512)+
....your normal program goes here

65ØØØ GOSUB 655ØØ+
.......Print out all the variables of interest
.......and any other debugging you desire
.......exit might be
65Ø9Ø GOTO 6529Ø+
651ØØ GOSUB 655ØØ: 'Save scrn+
......Debugging routine number 2
6519Ø.GOTO 6529Ø+
652ØØ GOSUB 655ØØ+
......Debugging routine number 3
6529Ø #MOV VARPTR(CP%(1)),1536Ø,1Ø24 'Restore screen+
65295 WPOKE &H4Ø2Ø,CP%(Ø):ENDKEY+
655ØØ CP%(Ø)=WPEEK(&H4Ø2Ø) 'Save cursor Pos+
6551Ø #MOV 1536Ø,VARPTR(CP%(1)),1Ø24  'Save the screen+
65515 RETURN+
```

(2) Saving DATA:

```
5 GOTO 1Ø+
7 SAVE"program":END+
9 PUT fan,&:ENDKEY+
1Ø ......+
1ØØ SETKEY8,CHR$(196) GOSUB 9      '<^ d>+
```

(3) MENU functions:

```
1ØØ CLS:PRINT"


               Menu 1 options


                                        ";CHR$(14);+
11Ø DO:A$=INKEY$:UNTIL A$=valid character(s)+
12Ø L2=12Ø: ON function(A$) GOSUB 2ØØØ,3ØØØ,4ØØØ,5ØØØ,14Ø+
13Ø GOTO 1ØØ+
14Ø CLOSE:CLS:PRINT@[1Ø,2Ø],"End of session.  Bye for now"+
15Ø END+
....
2ØØØ SK%(2)=KEYVAL4  'keep previous SETKEY assignment+
2Ø1Ø SETKEY 4,CHR$(27) GOSUB 2Ø6Ø   'set new SETKEY+
2Ø2Ø CLS:PRINT"

               Menu 2 options

                              ";CHR$(14);+
2Ø3Ø DO:A$=INKEY$:UNTIL A$=valid character+
2Ø4Ø ON function(A$) GOSUB 21ØØ,22ØØ,23ØØ,24ØØ,2Ø9Ø+
2Ø5Ø GOTO 2Ø2Ø+
2Ø6Ø SETKEY=N   'Disable all keys+
2Ø65 GOSUB 2Ø9Ø  'fix the files etc+
2Ø7Ø CMD"F=POPR",L2:RETURN+
2Ø9Ø close any open files+
2Ø93 SETKEY4,CHR$(27) GOSUB SK%(2)   'restore previous SETKEY+
2Ø95 RETURN+
```

Rather than continue on with an infinite depth of menus and routines, I think you can get the general idea from the above outline. Lines 1ØØ-15Ø is the main program loop. From here you can exit the program or enter other sub-menus. We've assumed control has passed to Menu 2.

The first thing done is to save the current SETKEY status (in case menu 2 was called from a menu other than menu 1) and the <shift up arr> set as the <escape> key. Regardless of where you may be, waiting at line 2Ø3Ø, or down in any of the sub-menus or routines available from menu 2: when the <shift up arrow> key is hit you will be diverted to line 2Ø6Ø. The code here has to make certain that all necessary housekeeping is done -- files closed etc.

The <CMD"F=POPR",L2> clears up the stack, (it also does the necessary resetting of SETKEY parameters on the way), and the RETURN will take you back to menu 1. If we replace the <CMD"POPR",L2> with <CMD"POPR",2Ø4Ø > then control would have been returned to menu 2 and exit back to menu 1 could be taken as an option.

A small program to finish to give you an idea of the priority concept of SETKEY.

```
1Ø DEFINT A-Z:PR$="   ":CLS+
2Ø SETKEY 1,"1" GOSUB 3ØØ+
3Ø SETKEY 2,"2" GOSUB 4ØØ+
4Ø SETKEY 3,"3" GOSUB 5ØØ+
5Ø PRINT"
   SETKEYS are assigned

         Hitting
                  1,2 or 3
                              will take you to the
                                 routine concerned."+
6Ø DO:PRINT@[9,Ø],CHR$(31):PRINT@[1Ø,2Ø],"Hit @ to end"+
7Ø GOSUB 7ØØØ:UNTIL A$="@"+
8Ø CLS:PRINT"End of demonstration":END+
3ØØ MID$(PR$,1,1)="X":GOSUB 3ØØØ+
31Ø MID$(PR$,1,1)=" ":ENDKEY+
4ØØ MID$(PR$,2,1)="Y":GOSUB 4ØØØ+
41Ø MID$(PR$,2,1)=" ":ENDKEY+
5ØØ MID$(PR$,3,1)="Z":GOSUB 5ØØØ+
51Ø MID$(PR$,3,1)=" ":ENDKEY+
3ØØØ PRINT@[9,Ø],"

          You have hit  1.
             Note that KEYS 1,2 and 3 now have no effect."+
3Ø1Ø GOTO 6ØØØ+
4ØØØ PRINT@[9,Ø],"

          You have hit  2.
             Note that KEYS 2 and 3 now have no effect
                but KEY 1 still works. "+
4Ø1Ø GOTO 6ØØØ+
5ØØØ PRINT@[9,Ø],"

          You have hit  3.
             Note that hitting 3 now has no effect
                but KEYS 1 and 2 still work. "+

5Ø1Ø GOTO 6ØØØ+
6ØØØ DO:PRINT@[15,3Ø],"Hit ";PR$;" to continue";+
6Ø1Ø GOSUB 7ØØØ:UNTIL INSTR(LSTRIP(PR$),A$)=1+
6Ø2Ø RETURN+
7ØØØ DO:A$=UPCASE$(INKEY$):UNTIL A$>"":RETURN+
```

-----------<ND86>-----------

PSSHT! SSSSH! WHISPER! WHISPER!

All 'good' programmers now that it is a sin (!?!) to exit any sort of control structure via a 'back door'. So please don't tell anyone, but I've extended the ND8Ø CMD"F=POPx" functions to include POPC, POPD and POPW. These will cause all current FOR-NEXT loops to be purged, along with the control structure specified. (Case, Do or While)

Program control then passes to the statement following the CMD"F=POPx". This allows an easy means of escape from nested control structures, without losing your variables or leaving residual control structures on BASIC's stack.

----------<ND86>----------

CMD"POPR",line #

This command allows you to purge all BASIC control structures (e.g. FOR...NEXT, DO...UNTIL, CASE...ENDCASE) until a GOSUB from the specified line number is found. A RETURN will cause execution to begin at the point following the GOSUB in the specified line. E.g.

```
1Ø GOSUB 1ØØ:IF A<>9 PRINT:PRINT"Exiting occurred via POPR+
2Ø END+
1ØØ FOR I = Ø TO 4+
11Ø PRINT "I=";I+
12Ø GOSUB 2ØØ+
13Ø PRINT:NEXT+
14Ø A=9:RETURN+
2ØØ A=Ø:PRINT"   A=";:DO+
21Ø A=A+1:PRINT A,+
22Ø IF INKEY$="Q" THEN CMD"F=POPR",1Ø:PRINT"exiting":RETURN+
23Ø UNTIL A=3+
24Ø RETURN+
```

While a simple example, it should convey the power of the command. See the section on SETKEY for other ways to initiate the POPR command.

----------<ND86>----------

CHAPTER THIRTEEN

GRAPHIC FUNCTIONS

CUSTOM BASIC provides several interesting graphic instructions. These may be used together with #MOV and the TONE generation routines as the basis for games, or by themselves as a means of generating particular screen formats quickly.

----------<ND86>----------

DRAWING LINES

```
#DRAW S  Ø,Ø TO 127,47
```

The format of this command is:-

```
#DRAW c [X1,Y1] TO X2,Y2 [TO X3,Y3 TO......Xn,Yn]
```

This command draws a line between X,Y and X1,Y1, where X and Y are screen co-ordinates within the normal range (Ø<=X<128 and Ø<=Y<48) of SET, RESET and POINT. Values beyond these limits will give an ILLEGAL FUNCTION CALL ERROR. If the optional first parameter set isn't used the line is started from the last XY set given. This defaults to Ø,Ø at initialisation. The 'c' refers to the function desired. Three functions are available with #DRAW:-

c = S   SET's the pixels between the specified points.

c = R   RESET's the pixels on the line specified.

c = X   XOR's the pixels on the line specified. That is, if the point is SET it will be RESET, and if RESET it will be SET.

Thus the statement:-

```
#DRAW S Ø,Ø TO Ø,47 TO 127,47 TO 127,Ø TO 1,Ø
```

will cause a line to be drawn around the perimeter of the screen. If an 'R' is substituted for the 'S', and the statement repeated, the line will disappear.

The 'X' option will cause the line to appear and disappear alternately. The 'X' option will not destroy any text as it won't draw any lines through text.

----------<ND86>----------

PLOTTING SHAPES

#PLOT A%(Ø) @ (64,24)

The other function offered is the ability to display shapes, stored in arrays, on the screen. The format for the command is:-

#PLOTc var%(Ø) [@ (X,Y)] ,size], rotation] [&...]

Obviously, the array must be specified, but the other parameters are optional. Once again 'c' specifies the action type of the routine:-

c = S causes the pixels to be SET

c = R RESETS the pixels

All parameters must be within INTEGER range (-32768 to +32767), otherwise an OVERFLOW ERROR is displayed.

The expression (X,Y) specifies the position of the first pixel to be acted upon, the normal screen co-ordinates of Ø<=X<128 and Ø<=Y<48 apply. However, if parameters are specified outside of these limits, no error messages are generated.

Rather, the routine will try to plot the shape at that specified position. Before any pixels are SET or RESET the routine tests to see if the specified pixel is within the screen area. If the co-ordinates are outside the limits, no action is taken. The co-ordinates are still changed, allowing the shape to be made to gradually come onto, or leave, the screen.

Varying the value of 'size' allows one to expand the shape to any size, from one to two hundred and fifty-six times normal. (Practical range of values is 1 to 6.)

The values of X,Y and size are modulo 256. This means that:-

Ø,1,2 = 256,257,258 = 512,513,514.... to INTEGER limits

There are 8 possible angles of rotation, i.e. forty-five degrees apart, in a clockwise direction. These are specified by the values Ø through 7. This value has a modulus of 8, that is:-

Ø,1,2 = 8,9,1Ø = 16,17,18.... to INTEGER limits

If we wanted to turn an object upside down we would rotate it through 18Ø degrees, therefore rotation would equal 4, as the following table shows:

ROTATION in a CLOCKWISE DIRECTION

VALUE	DEGREES
Ø	none
1	45
2	9Ø
3	135
4	18Ø
5	225
6	27Ø
7	315
8	none

Another option available is the ability to plot several shapes within the same command. This is accomplished by using the '&' character, followed by the next set of parameters. If two shapes, one in A%(Ø) and the other in B%(Ø), are desired to be plotted successively, the command:-

```
#PLOTS A%(Ø) @ (X,Y),1,Ø & B%(Ø) @ (X1,Y1)
```

will plot both shapes.

A point to note is that the routine stores all optional parameters permanently, and the values default to the last entered parameters. Thus, once size and rotation are set for one shape they remain in force until changed. The X,Y parameters, while retained, are altered by the routine, and the default values are equal to the last adjustment made. This aids in the "stringing" together of shapes. The original default values are Ø,Ø.

The following information may look somewhat daunting, but if you go through it step by step it really is quite simple to do. The BASIC program PLOTMATH/BAS will do the required maths, but it is limited to a 6Ø-move shape. Please read the following, if only for background information. Have fun with #PLOT, #REV and #DRAW.

The shape table is quite simple and comprises contiguous groups of five bits, each five-bit group is called a "cell"; see diagram below.

1	2	3	4	5
	direction		action flag	repeat flag

The previous cell is called an "action" cell. Bits 1 to 3 give the direction of movement, the values are shown in the following table. Bit 4, if set, indicates that the position is only to be changed without the pixel being affected. Otherwise the pixel is set or reset and the co-ordinates adjusted.

Setting bit 5 allows a particular cell to be repeated a number of times. If repeat is flagged, the following cell, called a "repeat" cell, defines the number of repeats. This allows lines to be drawn with just two cells (limit is 34 repeats). A repeat cell containing all zeros indicates a repeat of 3, since a repeat of 2 can be gained by using the same action cell twice. So when computing the value of the repeat cell, subtract 3 from the desired number of repeats.

The simplest approach to compiling the shape table is to view the screen as one would a map, i.e. North is up/South is down; East is right/West is left. Diagonal movement is also allowed. The bit pattern is shown below:-

DIRECTION	BIT VALUES		
N	Ø	Ø	Ø
NE	Ø	Ø	1
E	Ø	1	Ø
SE	Ø	1	1
S	1	Ø	Ø
SW	1	Ø	1
W	1	1	Ø
NW	1	1	1

As an example, assume that it is desired to draw a rectangle, seven pixels high by five pixels wide. Remember that the pixel is first acted upon, and then the co-ordinates are adjusted -- therefore, to have seven pixels set, only six movements are necessary, as the seventh pixel will be set by the following cell of the bit map. The order of movement is entirely arbitrary, so the rectangle will be built by a North-West-South-East combination. The shape table would be compiled as follows:-

MOVE	DIR	ACTION	REPEAT	NO. REPEATS
North * 6	ØØØ	Ø	1	ØØØ11
West * 4	11Ø	Ø	1	ØØØØ1
South * 6	1ØØ	Ø	1	ØØØ11
East * 4	Ø1Ø	Ø	1	ØØØØ1

Notice that we have subtracted 3 from all of the repeat lengths before we wrote down their binary equivalents in the VALUE column -- 6 becomes 3; 4 becomes 1. The binary values are now put side by side thus:-

North 6-3	West 4-3	South 6-3	East 4-1
ØØØØ1 ØØØ11	11ØØ1 ØØØØ1	1ØØØ1 ØØØ11	Ø1ØØ1 ØØØØ1

Now combine the ones and zeros into groups of four to translate to HEX values. It may be necessary to add trailing zeros to make up the final byte.

These values can then be loaded into an array (INTEGER arrays are the easiest ones to use). The first byte of the array holds the number of moves (lines) in the shape:-

MOVES	ØØØØ/1ØØØ	1111/ØØ1Ø ØØØ1/1ØØØ	1ØØØ/11Ø1 ØØ1Ø/ØØØ1
Ø4	Ø 8	F 2 1 8	8 D 2 1

Allowing for the LSB/MSB format of numeric storage, the INTEGER array, A%(x), should have the values:-

A%(Ø) = &H Ø8Ø4 = 2Ø52
A%(1) = &H 18F2 = 6386
A%(2) = &H 218D = 8589

Whilst this may appear difficult, in practice it is relatively simple. A simple BASIC program to use the above shape appears on the next page.

Regardless of the shape's complexity the shape should be plotted on graph paper (or better still, some TANDY stores have pads of paper designed for use with '8Ø graphics), the starting point can then be decided upon and the shape table compiled, as above.

Some sample programs and then a problem for you to try:-

```
10      CLS:CLEAR 500:DEFINT A-Z:DIM A(2)
20      FOR I=0 TO 2:READ A(I):NEXT
30      #PLOTS A(0) @(20,24)
40      FOR I=0 TO 400:NEXT
50      FOR L=0 TO 7
60      #PLOTS A(0) @(64,24),1,L
70      NEXT
80      FOR I=0 TO 400:NEXT
90      FOR H=1 TO 6
100     #PLOTS A(0) @(105,44),H,0
110     NEXT
120     #PLOTS A(0) @(20,24),1,0
130     GOTO 130
140     DATA 2052,6386,8589
```

This second example was submitted by Bill Allen:-

```
10      DEFINT A-Z:DIM A(5):CLS
20      FOR I=0 TO 5:READ A(I):NEXT
30      #PLOTS A(0) @ (60,18)
40      FOR I=0 TO 2000:NEXT:CLS
50      FOR X= -18 TO 140 STEP 10
60      FOR I= 0 TO 8
70      #PLOTS A(0) @ (X,24),1,I
80      NEXT I:NEXT X
90      CLS:GOTO 50
100     DATA 18440, 12490,-23673,-5432,-31432, 163
```

A simple problem for you to do. Whilst no working is shown, the answers for each stage are given. The problem:- Design a letter 'X' that is seven pixels high.

NE 7-3 00101 00100	WEST 7-3 11011 00100	SOUTH 10010	SE 7-3 01101 00100

MOVES 04	0010/1001 2 9	0011/0110 0100/1001 3 6 4 9	0011/0100 100*/**** 3 4 8 0

The array values are:-

```
A%(Ø)  =  &H 29Ø4      =  1Ø5ØØ
A%(1)  =  &H 4936      =  18742
A%(2)  =  &H 8Ø34      = -32716
```

Did you remember to pad the rightmost character with trailing zeros to make up a full HEX character?}

-----------<ND86>----------

REVERSE ON SCREEN GRAPHICS

#REV

This causes the graphic contents of the screen to be reversed while leaving any ASCII text alone.

```
1Ø     CLS
2Ø     #DRAWS Ø,Ø TO 127,47 TO Ø,47 TO 127,Ø
        TO Ø,Ø TO Ø,47 TO 127,Ø TO 127,47
3Ø     FOR I= 1 TO 3ØØ:NEXT
4Ø     #REV:GOTO 3Ø
```

-----------<ND86>----------

APPENDIX A

GENERATING SOUND WITH YOUR COMPUTER

CUSTOM BASIC provides a sound routine (BEEP). The output signal is conveyed to the real world via the cable that you use while saving your programs to tape (on most earlier models). The power level of this signal is very low and therefore you will require some form of audio amplifier to boost the level. Owners of the SYSTEM-8Ø BLUE LABEL and the Model 4/4P can skip this section since they have an internal amplifier already provided. For us lesser mortals, a few thoughts on how to get sound.

Some early tape recorders allowed you to monitor what was going onto the tape while recording. Such a tape recorder can be used as an amplifier. Set up the recorder as though you were going to CSAVE a program. Remove the tape from the recorder. If you look in the left rear corner of the cassette compartment, you will see a small lever. This lever prevents recording on a tape that has the write-protect notch removed. Lightly push the lever towards the back of the recorder and simultaneously depress both the record and play controls, locking the machine into record mode. Do whatever else is required to monitor the signal and you have sound.

Some of the later recorders that have built-in microphones won't let you monitor what you are recording. If this is the case you will need to use an external amplifier. Examine your video monitor for a 3.5mm socket, if it has one this normally indicates that an amplifier is provided. Otherwise you will either have to forgo the sound routines or find a small amplifier from another source.

----------<ND86>----------

APPENDIX B

SELF-BOOTING SYSTEM DISKS FOR THE 4P

For you Model 4P owners out there, here is an easy mechanism to create a self-booting ND86 disk from your WSS1 master.

1) Firstly do a LOGON and inform the system of your drive configuration, 4Ø or 8Ø track, single or double sided.

2) Type in COPY + /sys <enter>
Most of the prompts you should answer according to your needs with the following exceptions:-

Format Diskette? <Y>es or <N>o ? Y
The disk must be formatted if you wish to make a SYSTEM disk.

Is it to be a NewDos86 SYSTEM disk <Y>es or <N>o ? N
Tell it a lie. A self-booting disk must be in LDOS format.

Is the copy - <E>ntire disk <C>lone <F>ile F
It must be a file copy for the process to work.

This copies over the SYSTEM files only. The next step is to copy over a MODELA/III file. This should give you a self-booting disk.

SETTING UP ON EXOTIC DRIVES CONFIGURATIONS

To assist users with expanded drive systems though any PDRIVE hangups, the following leads you through the steps to take full advantage of your particular configuration. Make sure you copy the file COPY/ILF on to the system disk you are using.

CONFIGURATION ONE

Model I

DRIVE Ø = double sided 4Ø track drive.
DRIVE 1 = double sided 8Ø track drive.

Step 1:
PDRIVE Ø 1 TI=AL,TD=A,TC=4Ø,SPT=1Ø,TSR=3,GPL=2,DDSL=17,DDGA=2
PDRIVE Ø 5=1,TI=A,A
COPY 1,Ø,,CBF,FMT,DPDN=5
WSS is the SOURCE diskette while a blank disk is the DESTINATION diskette. Once the backup has been done, put the WSS disk away. Label the backup as WSS1.

Step 2:
COPY 1,Ø,,CBF,NFMT,ILF=COPY/ILF,DPDN=5
the DESTINATION diskette is WSS1, while the SOURCE diskette is your original NEWDOS8Ø master.

Step 3:
Remove all disks from all drives. Put WSS1 in drive Ø and re-boot.

BASIC,-,RUN"PATCH/BAS
and answer the prompts. After re-boot do

Step 4 for single density working:
PDRIVE Ø,1,TI=A,TD=C,TC=8Ø,SPT=2Ø,SPG=5,TSR=3,GPL=4,DDSL=35,DDGA=3
PDRIVE Ø,8=1,TC=4Ø,GPL=2,DDSL=17,DDGA=3
At this point, put a write protect sticker on WSS1 and re-boot.
Step 4 for double density working:
PDRIVE Ø,1,TI=C,TD=G,TC=8Ø,SPT=36,SPG=5,TSR=3,GPL=8,DDSL=35,DDGA=6
PDRIVE Ø,8=1,TI=CK,TC=4Ø,GPL=2,DDSL=17,DDGA=3
At this point, put a write protect sticker on WSS1 and re-boot.
Step 5:
SETSYS 1
when the program prompts you, take WSS1 out of drive Ø and put it in drive 1 and then hit <enter>. To make your working copy, put a blank disk in drive Ø and enter the command:-

COPY 1,Ø,,CBF,FMT,DPDN=8,E,NDMW
you now have a double sided 4Ø track master ND86 diskette.

CONFIGURATION TWO

Model I

DRIVE Ø = double sided 4Ø track drive.
DRIVE 1 = double sided 4Ø track drive.

Step 1:
PDRIVE Ø 1 TI=A,TD=A,TC=4Ø,SPT=1Ø,TSR=3,GPL=2,DDSL=17,DDGA=2
PDRIVE Ø 5=1,A
COPY 1,Ø,,CBF,FMT,DPDN=5
WSS is the SOURCE diskette while a blank disk is the DESTINATION diskette. Once the backup has been done, put the WSS disk away. Label the backup as WSS1.

Step 2:
COPY 1,Ø,,CBF,NFMT,ILF=COPY/ILF,DPDN=5
the DESTINATION diskette is WSS1, while the SOURCE diskette is your original NEWDOS8Ø master.

Step 3:
Remove all disks from all drives. Put WSS1 in drive Ø and re-boot.

BASIC,-,RUN"PATCH/BAS
and answer the prompts. After re-boot do

Step 4 for single density working:
PDRIVE Ø,1,TI=A,TD=C,TC=4Ø,SPT=2Ø,SPG=5,TSR=3,GPL=2,DDSL=17,DDGA=3
PDRIVE Ø,8=1,A
At this point, put a write protect sticker on WSS1 and re-boot.
Step 4 for double density working:
PDRIVE Ø,1,TI=C,TD=G,TC=4Ø,SPT=36,SPG=5,TSR=3,GPL=4,DDSL=17,DDGA=3
PDRIVE Ø,8=1,TI=CK
At this point, put a write protect sticker on WSS1 and re-boot.
Step 5:
SETSYS 1
when the program prompts you, take WSS1 out of drive Ø and put it in drive 1 and then hit <enter>. To make your working copy, put a blank disk in drive Ø and enter the command:-

COPY Ø,1,,CBF,FMT,DPDN=8,E,NDMW
you now have a double sided 4Ø track master ND86 diskette.

CONFIGURATION THREE

Model III

DRIVE Ø = double sided 4Ø track drive.
DRIVE 1 = double sided 8Ø track drive.

Step 1:
PDRIVE Ø 1 TI=AL,TD=E,TC=4Ø,SPT=18,TSR=3,GPL=2,DDSL=17,DDGA=2
PDRIVE Ø 5=1,TI=A,A
COPY 1,Ø,,CBF,FMT,DPDN=5
WSS is the SOURCE diskette while a blank disk is the DESTINATION diskette. Once the backup has been done, put the WSS disk away. Label the backup as WSS1.

Step 2:
COPY 1,Ø,,CBF,NFMT,ILF=COPY/ILF,DPDN=5
the DESTINATION diskette is WSS1, while the SOURCE diskette is your original NEWDOS8Ø master.

Step 3:
Remove all disks from all drives. Put WSS1 in drive Ø and re-boot.

BASIC,-,RUN"PATCH/BAS
and answer the prompts. After re-boot do

Step 4:
PDRIVE Ø,1,TI=A,TD=G,TC=8Ø,SPT=36,SPG=5,TSR=3,GPL=8,DDSL=35,DDGA=6
PDRIVE Ø,8=1,TI=A,TC=4Ø,GPL=2,DDSL=17,DDGA=3
At this point, put a write protect sticker on WSS1 and re-boot.
Step 5:
SETSYS 1
when the program prompts you, take WSS1 out of drive Ø and put it in drive 1 and then hit <enter>. To make your working copy, put a blank disk in drive Ø and enter the command:-

COPY 1,Ø,,CBF,FMT,DPDN=8,E,NDMW
you now have a double sided 4Ø track master ND86 diskette.

----------<ND86>----------

APPENDIX C

NUMBER BASE CONVERSIONS

This section is not meant to be a definitive work on the subject. For that I suggest you visit your local library. The primary purpose of this appendix is to help in the conversion of binary numbers to their decimal or hex equivalents, particularly in connection with the #PLOT routine.

binary	octal	decimal	hexadecimal
ØØØØ	Ø	Ø	Ø
ØØØ1	1	1	1
ØØ1Ø	2	2	2
ØØ11	3	3	3
Ø1ØØ	4	4	4
Ø1Ø1	5	5	5
Ø11Ø	6	6	6
Ø111	7	7	7
1ØØØ	1Ø	8	8
1ØØ1	11	9	9
1Ø1Ø	12	1Ø	A
1Ø11	13	11	B
11ØØ	14	12	C
11Ø1	15	13	D
111Ø	16	14	E
1111	17	15	F
1ØØØØ	2Ø	16	1Ø

The chart above is intended to help in this respect. Hexadecimal notation requires 16 'characters'. The numerals Ø through 9 give ten of the sixteen, the other six are gained by using the letters A-F.

Most of us can compute the value of a number without even thinking about how we arrive at the answer. To be able to work in any number base it is essential to understand how we do arrive at these answers.

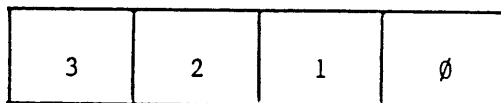
All of us are familiar with dealing with decimal numbers, i.e. numbers in base 1Ø format. Numbers in other bases can be, and are, just as easy to work with.

For simplicity we will limit our discussion to numbers of no more than 4 figures. Obviously, to work with larger figures will involve nothing more than extending the method. Characters are numbered away from the decimal point in a right to left format.

Our number systems hinge on the fact that the value of a numeral can change according to its position within the number. Thus the numerals 7 in the string, 7467, indicate two differing values. The first 7 (on the right) has a value of only 7 whilst the second 7 (on the left) has a value of 7ØØØ.

Let's examine why this is so for the base 1Ø system and then we can extend to other number bases, such as binary, octal or hexadecimal.

Firstly let's draw a rectangle and split it up into sections -- each section representing one numeral. Then number each section starting with zero, in the right to left format previously mentioned.



When I went to school it was explained that these numbers indicated how many zeros to put after the numeral that occupied the particular box. Thus

$$7467 = 7000 + 400 + 60 + 7$$

While this is true, it is only part of the answer. We could display the above equation somewhat differently.

$$7467 = 7 * 1000 + 4 * 100 + 6 * 10 + 7 * 1$$

This may help to make the relationship between the position of the numeral and the multiplier to use somewhat easier to see. Note that we are dealing with decimal or base 1Ø numbers.

$$1000 = 10^{3} \qquad 100 = 10^{2} \qquad 10 = 10^{1} \qquad 1 = 10^{0}$$

It becomes apparent that the value of the multiplier is the number base raised to the power of the position in the string. (Note that any base raised to the power of zero always equals one.)

When we are dealing with a different base, the same rule applies, but we end up with a different multiplier value. The same number string in octal or base 8 would gives us

$$7467 = 7 * 8^{3} + 4 * 8^{2} + 6 * 8^{1} + 7 * 8^{0}$$

This equals 3895 decimal. A little test problem: What would 7467 hexadecimal be in decimal? Answer below.

Base 2 or binary is a little easier as the highest possible numeral is '1'. The number string 1Ø11 binary would have the decimal value

$$1\emptyset11 = 1 * 2^{3} + \emptyset * 2^{2} + 1 * 2^{1} + 1 * 2^{\emptyset}$$

which equals 11 decimal.

Up to now we have been converting from other bases to decimal, to convert in the opposite direction is just as easy. Let's express 3895 decimal in octal. Just as going from octal to decimal required repeated multiplication, so going from decimal to octal will require repeated division.

The technique is to keep dividing the number by the base we want to convert to and storing the remainders. When the answer becomes Ø the last remainder is the final character of the new number. It is easier to do it than to explain it:-

```
   !
 8 !   3895                     remainders
   ------------
     !
     !  486  --------------------------------> 7
     ------------
       !
       !  6Ø  ------------------------> 6
       ------------
         !
         !   7  ----------------> 4
         ------------

             Ø     --------> 7
                         ===========================

                              7     4     6     7

                         ===========================
```

The same method can be applied to decimal to hex, in which case we would divide by 16. Remember also that remainders > 9 would be converted to the characters A-F. The table on page C-1 will help you. (Answer to problem is 29799 decimal)

-----------<ND86>----------

TOKENISED KEYWORDS IN BASIC

Name	Value	Name	Value	Name	Value
'	251	ERR	195	OR	211
#	207	ERROR	158	OUT	160
+	205	EXP	224	PEEK	229
-	206	FIELD	163	POINT	198
/	208	FIX	242	POKE	177
<	214	FN	190	POS	220
=	213	FOR	129	PRINT	178
>	212	FRE	218	PUT	165
ABS	217	GET	164	RANDOM	134
AND	210	GOSUB	145	READ	139
ASC	246	GOTO	141	REM	147
ATN	228	IF	143	RESET	130
AUTO	183	INKEY$	201	RESTORE	144
CDBL	241	INP	219	RESUME	159
CHR$	247	INPUT	137	RETURN	146
CINT	239	INSTR	197	RIGHT$	249
CLEAR	184	INT	216	RND	222
CLOAD	185	KILL	170	RSET	172
CLOSE	166	LEFT$	248	RUN	142
CLS	132	LEN	243	SAVE	173
CMD	133	LET	140	SET	131
CONT	179	LINE	156	SGN	215
COS	225	LIST	180	SIN	226
CSAVE	186	LLIST	181	SQR	221
CSNG	240	LOAD	167	STEP	204
CVD	232	LOC	234	STOP	148
CVI	230	LOF	235	STR$	244
CVS	231	LOG	223	STRING$	196
DATA	136	LPRINT	175	SYSTEM	174
DEF	176	LSET	171	TAB(	188
DEFDBL	155	MEM	200	TAN	227
DEFINT	153	MERGE	168	THEN	202
DEFSNG	154	MID$	250	TIME$	199
DEFSTR	152	MKD$	238	TO	189
DELETE	182	MKI$	236	TROFF	151
DIM	138	MKS$	237	TRON	150
EDIT	157	NAME	169	USING	191
ELSE	149	NEW	187	USR	193
END	128	NEXT	135	VAL	245
EOF	233	NOT	203	VARPTR	192
ERL	194	OPEN	162	[	209

----------<ND86>----------

THE CUSTOM BASIC TOKENISED KEYWORDS

CUSTOM BASIC now tokenises all of its keywords. The tokenisation is accomplished in the following fashion. The routine 'piggy backs' the CUSTOM BASIC tokens onto a standard BASIC token. The MID$ token is used for commands and the TIME$ token used for functions.
The BASIC token is followed by an ØFFH byte to indicate that it is not a 'normal' BASIC token. Following the ØFFH is a byte which identifies which CUSTOM BASIC routine is being accessed. What follows is the table of CUSTOM BASIC tokens. The leading BASIC token+ØFFH byte sequence is not shown.

Name	Value	Name	Value	Name	Value
;	7	FILL	28	RSTRIP(	33
AND$	38	FCB	43	SET	47
BEEP	1	INC	5Ø	SETKEY	39
CALL	25	INSTR(!	19	SUM	31
CASE	6	KEYVAL	41	#SWP	14
CHR$64	9	KTA	45	$SWP	22
DEC	51	$LET	27	UNTIL	24
DEFUSR	29	LSTRIP(	32	UPCASE$(	8
#DO	2	MOD(	42	USR	34
DO	23	#MOV	12	VAR	15
#DRAW	3	OR$	36	VID(	11
#GOTO	4	#PLOT	13	WAIT	46
EDIT	18	POINT	49	WEND	21
ENDCASE	5	RESET	48	WHILE	2Ø
ENDKEY	4Ø	#RESTORE	1Ø	WPEEK	17
ERR$(	3Ø	#REV	26	WPOKE	16
ERROR	35	ROT$	44	XOR	37

----------<ND86>----------

=============== NEWDOS86 (C) 1986 W.S. & D.S. SANDS ===============

ORIGINAL CUSTOM BASIC COMMANDS AS APPLICABLE TO TAPE USERS

What follows is a syntax list of the original CUSTOM BASIC routines as applicable to tape users. The routines that are not self-explanatory and are no longer included in the manual have a brief explanation. At the end is a brief resume of CUSTOM BASIC for TAPE USERS.

```
GENERAL ROUTINES
    &H hex$, &D decimal exp and &% decimal exp
    BOOT
         causes a reset of the computer
    CHR$64
    #GOTO line
    #RESTORE line
    WPOKE address,value

MOVE MEMORY BLOCKS
    #MOV from, to, bytes to move
    #SWP from, to, bytes to move

TONE GENERATION
    BEEP tone, duration: BEEP exp$
    BUZZ duration
         turns the cassette relay on and off
         to make a buzzing sound.

STRING FUNCTIONS
    CVI(string), MKI(string), MKD(string)
    #DO exp$
    FIELD @ address, var [ LEN exp ]
    var = INSTR( [starting position,] searched$, search$)
    INPUT TO [@pos,] ["prompt string";] var1$ [*] [;] USING var2$
    LINE INPUT [prompt string;] var$
    MID$ FUNCTION
    MKI$(value), MKS$(value), MKD$(value)
    $SORT  var$(start),number of items
         Sorts a singly dimensioned string array.

USER DEFINED FUNCTION
    DEF FN var [(parameter list)] = function definition
    var = FN var [(argument list)]
    DEFUSR Ø-9 = address of routine
1Ø USR FUNCTIONS
    var = USR Ø-9(argument)
    DEFUSR =

ON-SCREEN GRAPHICS
    #DRAWc [x1,y1] TO x2,y2 [TO x3, y3...]
    #PLOTc var%(Ø) [@(x1,y1)] ,size] ,rotation]
```

TAPE FUNCTIONS

LINE INPUT [#-exp,] var$
Inputs data from tape

LOAD [#-expl,] last line to retain [,name$]
loads a BASIC program overlay - variables saved

PRINT* [#-expl,] [*(number of leader bytes);] DATA
Sends data to tape with a reduced leader

SYSTEM tape FUNCTIONS

SAVE SYSTEM [#-exp,] name$, entry address, startl,endl....

LOAD [#-expl,] SYSTEM name

BASIC tape FUNCTIONS

MERGE

MERGE [#-expl,] last line to retain [,name$]
Merges two programs - variables saved

SAVE linel TO line2, name$
Saves portion of a BASIC program to tape

UTILITIES

LMOV [C] linel [:line2] TO line3
Same as DI-DU

REN [start],inc],first line],last line]

RENEW
restores BASIC program

VAR [L]
lists simple variables only

EDIT DIRECT COMMAND STATEMENTS
LOWER CASE DRIVER
SINGLE-KEY ENTRY
AUTO REPEAT/FLASHING CURSOR

PRINTER FUNCTIONS

DUMP SCREEN TO PRINTER

LPRINT SET/RESET

IN-MEMORY SOFTWARE SPOOLER

MARGIN ROUTINE

SERIAL PRINTER DRIVER
Needs a hardware add-on

EXTENDED BASIC was designed to RUN on TRS-8Ø MODEL I, MODEL III or SYSTEM 8Ø computers, with 16K/32K/48K memory capability with the program itself compensating for any differences between machines.

EXTENDED BASIC FOR '8Ø USERS comprises sixteen BASIC program modules which will compile into memory a powerful add-on machine language program, CUSTOM BASIC FOR '8Ø USERS.

The advantage of EXTENDED BASIC is that you, the user, can select precisely what routines you want for a particular application and create a SYSTEM tape having only those routines.

Routines include most DISK BASIC commands (excepting Disk I/O), such as LSET, RSET, INSTR, MID$=, CVI-MKI$, LINEINPUT, DEF USR, DEF FUNCTION, FIELD@. As well as being able to convert data from HEX to decimal you can also convert DATA from DECIMAL to HEX.

Also included is the ability to dynamically append BASIC programs, thus allowing a CHAIN command. It is also possible to do a true MERGE from tape of two BASIC programs. A side effect is the ability to edit a BASIC program and retain your variables.

It is also possible to CSAVE portions only of your BASIC program to tape.

The are several printer functions. An in-memory spooler, that stores the printer data and transmits it to the printer only when the printer is ready.

The ability to dump the screen to the printer, either from the keyboard or from within your program.

The <break> key will exit you from an LLIST to a non-existent printer, (same as MOD III).

You can also cause all data typed at the keyboard or printed to the screen to go to the printer. This also can be initiated from either the keyboard or from within the BASIC program.

Finally, there is a provision to set both left and right and top and bottom margins for your printer, thus allowing a certain degree of formatting your listings. It will break words and lines however.

Keyboard routines include auto-repeat, flashing cursor, lower case drivers, single key entry of tokens, sound with keystrokes.

You can also EDIT statements typed in command mode. You can use this to duplicate program lines in memory, or make a command into a program line.

Other miscellaneous commands include the ability to reduce the length of the leader when PRINT#'ing data to tape. You can also do a LINEINPUT from tape.

I have included a powerful new routine to INPUT data from the keyboard, this overcomes all of the deficiencies of BASIC's INPUT routine.

It is possible to dump multiple areas of memory to tape and reload them from tape from within a BASIC program as well as from command mode.

Also included are a fast string sort for single dimension arrays. 1ØØØ items in approx 7 seconds. The ability to partially restore DATA statements. You can GOTO a line specified by a numeric expression. The ability to move and swap memory is provided. Two graphic routines are provided. The ability to get BASIC to treat a string expression as a direct command.

You can move and duplicate program lines in memory, a renumber routine is also provided. You can also list out all the non-array variables and their contents to either screen or printer, handy for debugging.

USER GROUPS SUPPORTING NewDos86

We do hope that this list will grow. What does supporting NewDos86 involve for a user group? Not a real lot actually as the following list shows.

1) Be willing to print any zaps to ND86 in its newsletter as these are made available.

2) Be willing to print articles about ND86 in its newsletter, if the editor feels that these would be of interest to the members of the club.

3) Be willing to report back any problems that might arise so I can fix them.

In short, to provide a forum for discussion on subjects relating to ND86. If the club wishes to act as an official distributor of ND86 that would be fine, but it is not a requirement. If you are interested in either distributing or just supporting ND86 please contact me via one of the Australian Groups.

In AUSTRALIA

The TRS-8Ø SYSTEM 8Ø Computer Group,
General Secretary,
41 Montclair St,
Aspley,
Queensland,
Australia, 4Ø36

In EUROPE

National Amstrad, Tandy & General User Group
D. Washford,
6 Houston Way,
Frome,
BA11-3EU,
SOMERSET, U.K.

NOTES FOR PURCHASERS OF HARDOS86 HARD DISK SUPPLEMENT

BOOT THE SUPPLIED DISK AND MAKE A COPY OF IT.
NOW ISSUE THE COMMAND:-

BASIC RUN"PATCH/BAS

The file HDFMTAPP/CMD is the original 2.5 low level formatting utility modified to enable handling the WD1000 series hard disk controllers as used on the later tandy hard drives. It is still ok to use on the old 8X600 controllers. If your drive is already low level formatted, there's no useful purpose served in doing it again.

The following are the steps required to make a booting 2.5H system. Assume that the floppy drive is drive :0 and the Hard drive partition we are going to make the hard drive system is Drive :2.

The PDRIVE parameters on this disk have been set to suit a Tandy 15 meg drive -- slots 2 - 7 in 6 volumes with the first volume (:2) being much smaller as it only needs to accommodate the system and other enhancements. For further information on PDRIVE settings you need to read Newdos80 2.5's Appendix C.

If the supplied PDRIVE parameters do not suit your purpose, you must set the PDRIVE parameters for slot 2 to recognise the hard disk partition that is to contain the NewDos86 system. I most earnestly suggest that you set the SPG parameter of this volume to at least 6. In the future I hope to release some
upgraded hard disk /SYS files, and these will be 6 sectors in length. The supplied settings already suit.

Copy the supplied floppy to the Hard Drive with a

COPY,0,2,,CBF,FMT

If the partition contains data, back this up as the copy process will erase all data on that partition.

Next enter SUPERZAP use CDS from the menu

```
Responses: Y          (do you really want to do this?)
           2,2        (source and relative sector)
           2,3        (destination and relative sector)
           1          (sector count)
```

and do the normal exit bit.

To make the floppy boot disk:-
Put a blank floppy in drive :1 and format it, Copy across SYS0/SYS:2, and BOOT/SYS:2.

To make a self-booting disk that also loads the MODELA/III file (for a 4P) the following steps must be taken.

Using ND86v2.0 (not hard disk). Set up a PDRIVE SLOT as follows:-

PDRIVE,0,8,TI=AN,TD=E,TC=40,SPT=18,SPG=6,GPL=3,DDSL=20,DDGA=3

mount a blank disk in drive 1 and the ND86 4.5H in drive 2 and

COPY 2,1,,CBF,Y,DPDN=8

This sets up the disk so that the BOOT/ROM can find the MODELA/III file. Since HD86v2.5 stores the SYSTEM option in sector 3 of BOOT/SYS and since ND86v2.0 doesn't update sector 3, you will need to enter DEBUG and type:

L1,2
WR1,3

This updates sector 3 correctly. This fudge is a temporary one and will soon be unnecessary.

Change the PDRIVE value for slot 0 to suit the hard drive and it should boot.

To make copies of the BOOTing disk you will need to use ND86v2.0 (not hard) and use the command:

COPY x,y,,SPDN=8,DPDN=8,BDU

this will give you identical copies.

-----------<HD86>-----------

Some of the known limitations of HarDos86v2.5.

COPY and FORMAT have not been enhanced. (YET!)
This means that there may be some unwanted interaction between the Automatic Disk Format Determination Routine and FORMAT. For the moment, when formatting a Floppy, specify a DPDN parameter.

Whenever a SYSTEM disk is created, the ND86 SYSTEM options will most definitely be incorrect.

Some of the files mentioned below may not be supplied as they do not work with the Hard disk version of NewDos86. If you find a problem with any of these utility files please let me know and I will try to fix the problem.

Please note that there are other information files on the reverse side of this disk that you should print out and add to the manual.

-----------<HD86>-----------

NOTES FROM A USER ON SETTING UP A
TANDY 15 MEG DRIVE FOR HARDOS86

There are two basic methods of allocating the sectors on the hard disk. The first mechanism, usually used by LDOS/TRSDOS, tries to allocate a single volume to a single disk surface. The reason for doing this was twofold. Firstly it is easy to comprehend and secondly if a single head should become inoperative you will not lose all of your data. The disadvantage of this method is that more head movement will be involved during I/O operations.

The second formatting arrangement, as used by MS-DOS, is to allocate all available heads to each volume, rather than use a complete surface for a volume. This gives shorter and therefore on average faster movement between directory and files and therefore faster I/O and as a bonus less wear and tear on the stepper mechanism. The disadvantage is that if a head should crash, then the data that is affected will be spread across all the volumes.

Which method you use is entirely up to you. Appendix C as supplied with the NewDos80v2.5 disk gives many examples of the first method, nut hints that you should use the other method. Therefore we will discuss how to set up the hard disk using the alternative (MS-DOS) method.

When you are HDFMTAPPing ("hard" formatting), an interleave of 4 seems the best for this particular drive (despite what Appendix C states), 32 sectors/track, stepping rate of 15 (formatting only -- note I use 0 in the application usage). The recommendations in Appendix C refer only to the (early) 5 meg drives and these 15 meg models are capable of much better performance in stepping and the sector interleave.

Basically the initialising plan (if you are going to make it an all HD86 drive) is:
1. Copy the supplied disk and store the original away in a safe place.
    COPY 0,1,,FMT,BDU <ENTER>
2. Now put the copy in 0.
3. Execute HDFMTAPP using the tips above if the drive is not yet hard formatted (this will take up to 45 minutes). If already low level formatted, skip step 3.

PDRIVE SETUP
Assuming all 6 heads are available for HD86, the following is the most efficient plan for the PDRIVE table of Hard Drive 0 (15 meg) with 6 volumes of equal size of 2.5 meg:

```
0 *HDS=(0,306,0,6,32,0,0,9792,7,8,61,33)
1 *HDS=(0,306,0,6,32,0,9792,9792,7,8,61,33)
2 *HDS=(0,306,0,6,32,0,19584,9792,7,8,61,33)
3 *HDS=(0,306,0,6,32,0,29376,9792,7,8,61,33)
4 *TI=A,TD=E,TC=40,SPT=18,TSR=0,SPG=0,GPL=2,DDSL=17,DDGA=3
5 *TI=A,TD=E,TC=40,SPT=18,TSR=0,SPG=0,GPL=2,DDSL=17,DDGA=3
6 *HDS=(0,306,0,6,32,0,39168,9792,7,8,61,33)
7 *HDS=(0,306,0,6,32,0,48960,9792,7,8,61,33)
8  TI=A,TD=E,TC=40,SPT=18,TSR=0,SPG=0,GPL=2,DDSL=17,DDGA=3
9  TI=A,TD=E,TC=40,SPT=18,TSR=0,SPG=0,GPL=2,DDSL=17,DDGA=3
```

You'll have Hard Volumes 0,1,2,3 floppies 4,5 (0,1 at boot), and Hard Volumes 6,7. If desired (and it's a valid plan), you could make hard volume 0 (the system volume) much smaller by limiting its sectors to about 1152 (a precise figure for a system volume which would then only span 6 cylinders) and sharing the extras around the other data volumes by adjusting the 7th and 8th subparameters in the hard volume definitions. The only thing about making a relatively small system volume -- whatever you do, don't make the SPG (9th subparameter) less than 6, because Warwick intends to expand some of the system files to 6 sectors later on and with a minimum SPG=6 (larger than 6 is ok too) you will be configured correctly to incorporate the changes with very little drama to a working system. A smaller system volume could have a smaller directory (DDSA, the 12th subparameter) -- 15 sectors would be the recommended minimum, but it may need more according to what you plan to keep on the system volume. Don't forget that on hard drives one often runs out of directory space sooner than storage space. This means it also pays to configure the system to as many volumes as practicable.

You have to view the 15 meg physical drive as having 58752 sectors (this is caused by ALL the 3rd and 4th subparameters being 0 and 6) and the PDRIVE 7th and 8th subparameters of each definition allocate these sectors starting with 0. If changing the sectors allocation, you must be VERY SURE that your figures are correct -- just start with volume 0 and add on the 8th subparameter to the 7th to get the 7th subparameter in the next volume and so on. In the final volume, the 7th and 8th subparameters should add up to 58752 -- else you've made a boo-boo and will have to go back and check all the 7th and 8th subparameters. Just to confuse the issue a bit, you could also put the system drive sectors in the middle of the drive simply by some tricky juggling of the sector allocation subparameters -- they don't HAVE to follow on like little doggies down through the volume numbers. This may also help lessen head stepping on average between volumes. A further help would be to use the volumes farthest away from the system cylinders for storage of little used data or archived programs etc. with the commonly used data volumes closest to the system cylinders. A lot of thought and planning before you finalise the PDRIVE definitions will extend the life of your hard drive stepping mechanism quite considerably.

Once the drive has been fully initialised, it's just the same as addressing 8 floppies in the system, excepting that the Hard Volumes are lightning fast and have bags of room.
(The floppies HAVE to start at SLOT 4 in this system).

4. From Dos Ready, COPY 0,2,,FMT,CBF
5. FORMAT 3 (then 4, 5, 6, & 7 in turn -- this is the "soft" format by Dos)
6. Put a blank disk in 1 and FORMAT 1.
7. COPY SYS0/SYS:2 :1
8. COPY BOOT/SYS:2 :1 (this disk then becomes the Boot disk which will turn system control to the first Hard Volume which then becomes 0 and the others follow in accordance with the Hard Volume PDRIVE configuration).

If you have a split system, each will be invisible to the other. However, a temporary change to a PDRIVE for a complete surface on the "foreign" system to HARDOS86 will allow you to inspect the sectors (DTS function) with the hard drive version of SUPERZAP and alter them if needed. This SUPERZAP also does the normal functions on the floppies.

Once the system is up and running, I do a PROT 0,NAME=HARDA, PROT 1,NAME=HARDB, etc to give the hard volumes their identity in hard copy DIRs etc.

-----------<HD86>-----------

USING HARDOS86

Be aware that Newdos/80 2.5 uses a portion of high memory for the hard drive operating code. As NewDos86 also uses portion of high memory for its enhancements, some applications may find the total loss too much. If this happens, try booting with the Shift Key or F2 key held down and the high memory enhancments will not load. If there is insufficient room for a current Basic program, then you may install the ordinary Newdos/80 Basic from a Newdos/80 disk using OLDBASIC/ILF, instead of Custom Basic. It usually is beneficial to retain the AFDR high memory code to handle the different format disks. Usually there is plenty of memory for Basic and most applications programs with everything installed.

There is merit in considering a double system setup for special purposes where memory constraints intrude. First make your ordinary HARDOS86 setup, then copy another system to one of the other volumes and set it up with the ordinary Basic modules as above. You still need to copy that sector in the BOOT/SYS of that new volume and make a booting disk for that volume. After that is done, you may select the system you need by inserting the BOOT disk for the selected system and using it to BOOT.

-----------<HD86>-----------

HELPDISK

Helpdisk is a facility designed for use on NewDos8Ø/86 and is supplied on your WSS disk as a system file. It allows the user to interrupt almost any task or any mode on the computer, save the situation to return to later and then give access to text files anywhere in the system to look up any desired reference or information and return when desired without losing a single bit of the previous work being done. In short, this adds an encyclopedic function to your computer.

To make it very easy to use, HELPDISK is designed to be CURSOR driven as far as possible -- i.e. you are mainly using the arrow keys. Other designated keys are used for function only -- there is no typing done in Helpdisk.

Helpdisk can look at specially indexed files (/HLP files list below) where you can work from the index. It also can scan any ASCII text files. For proper usage, text should be formatted to 63-character width, with CHR$(13)s terminating each line to prevent word wrap around. Formatting your own files may be done by sending a print file to disk instead of printer with Lazywriter 3.5 or TYPE/CMD (on the public domain disk).

To enter HELPDISK, hold the 123 keys down FIRMLY until the screen flashes, then release them. If the SYSTEM option CH=N, then Helpdisk is accessed by the normal hold-down-123 method. If CH=Y, then Shift-123 accesses Helpdisk and 123 always gives you DEBUG.

You must NOT have a write-protect tab on the system disk when accessing HELPDISK. The Helpdisk driver saves the current screen and other necessary parts of RAM to a space within its file on the system disk so it can restore the situation when you exit. If you enter DEBUG accidentally, type G-<ENTER> and you're back to where you were just before you pressed 123 and you may try again.

Note that if you enter DEBUG, your screen will NOT be saved, as it is when you enter HELPDISK.

When you enter HELPDISK, it does the save and searches the default drive's directory for any /HLP files.

There is a small Basic utility that will change the default drive to DIR for /HLP files -- HELPDFLT/BAS)

These /HLP files, if any, will be printed to the screen, with the cursor placed at the start of the first /HLP file found. If you desire to search the other drives for /HLP files, just press the corresponding numeral key (e.g. <1> for Drive 1 etc). The cursor is positioned on ANY character of the desired helpfile and <ENTER> pressed.

The start of the file's index (if indexed) will now appear -- it can be scrolled up and down with UP and DOWN ARROWS until the cursor is placed on the desired index reference. For greater speed of scrolling, use <SHIFT> with the ARROWS. When the cursor is placed ANYWHERE on the desired word or phrase, press <ENTER> and the information will quickly appear on screen. The UP-DOWN ARROWS have the same action in the information area -- in fact, the index will scroll down if you pass the topmost reference information.

If you're not sure how to escape, press <I> for INSTRUCTIONS and they will appear instantly. Pressing any key will get you out of the instructions screen and back to where you called that screen. <H> for HELP does the same as <I>.

If you see some word (or character or word combination) you desire to reference while viewing an information screen, place the cursor anywhere on the word and press <ENTER> and the information will appear if there is indeed a reference for that word requested. If no reference exists in the file for that request, nothing will happen at all.

The <BREAK> key is used to step backwards through the reference screens called by the use of <ENTER>. This enables easy exploration of the information as though you were exploring a tree by going back to a fork and taking an alternate branch, twig etc. The depth of the stored reference path can be as much as 6Ø. It is very doubtful that this would be tested in normal usage!

Exit back to the original entry to HELPDISK is made from any point by pressing <CLEAR-BREAK>.

FULL FUNCTION KEYS LIST

<123> or <SHIFT-123> entry to HELPDISK.

The function keys used within HELPDISK are:

<ENTER> Reference the word/filename under the cursor, if such exists.
<BREAK> Go back to the previous reference screen.
<SHIFT-BREAK> Go directly to last DIR /HLP screen.
<I>nstructions or <H>elp -- either will print a screen of instructions.
<ARROWS> Move cursor in that direction and scroll when screen limit is reached.
<SHIFT-ARROWS> Much faster movement to cover ground quickly.
<R> Remember this spot in the text or index.
<B> Bring back the <R>emembered spot.
<@> Return to the index from any depth.
<S> takes you back to the Start of the file (same as <@>).
<E> takes you to the End of the file.
<M> takes you to the Middle of the file.
<U> skips 1Ø sectors Up (towards the Start).
<D> skips 1Ø sectors Down (towards the End).
<SHIFT-U> or <SHIFT-D> skips 4Ø sectors instead of 1Ø.
<;> advance 1 sector
<-> go back 1 sector
<CLEAR-BREAK> Restore original program or command mode.

CURRENT LIST OF /HLP FILES:

ND86/HLP (61 grans -- full instructions for the use of
    ND8Ø and ND86's LIB commands)
AIDSIII/HLP (2Ø grans)
DTWRITER/HLP (11 grans)
LAZYWRTR/HLP (92 grans)
MACASMON/HLP (38 grans -- available only for purchasers of
    Macasmon)
SUPERZAP/HLP (18 grans -- very comprehensive instructions)
VISICALC/HLP (14 grans)

OTHER FILES ASSOCIATED WITH HELPDISK

HELPDFLT/BAS (will change the default drive to seek /HLP files)
HELPINST/TXT (info on how to create your own /HLP files)
HELPND86/BAS (program to create /HLP files from prepared text)
HELP/HED (the graphics appended by creation utility)