Param (11) 2

Param(11) = 8
= 8 Digit Preci

Param(11) = 10
= 10 Digit Precision

Param (11) = 14
= 14 Digit Precision

REFERENCE MANUAL TO THE APCBASIC PROGRAMMING LANGUAGE

REVISION 2.4

****** TABLE OF CONTE**

**** 1.0 INTRODUCTION TO A

This manual is a reference guide to the facilities provided by APCBASIC creating, modifying, debugging and running programs written in the APCBASIC programming language. Since it is intended for programmer reference, rather than as a tutorial, people unfamiliar with general BASIC programming should select a beginning BASIC programming guide to supplement this manual for further clarification of BASIC structures. A working knowlege of your computer system and its operating system is assumed (especially its file handling capabilities).

Section 1.0 is written not only to introduce APCBASIC to the programmer, but to fully orient a person who will be running programs written by someone else and who is generally disinterested in the details of actual APCBASIC programming. It includes a full description of the APCBASIC line editor available whenever entering programs or data from the keyboard. Section 2.0 describes all commands available under the APCBASIC command level. Sections 3.0 through 7.0 explain the facilities supported by the APCBASIC language. Section 8.0 describes several programming aids which are included in the APCBASIC software package. Section 9.0 describes APCBASIC from the point of view of a North Star BASIC user moving up to APCBASIC, ie. as a super-set of that language.

The minimum system requirements for using APCBASIC include a Z80 microprocessor, 32K memory or more, at least one floppy disk drive (5.25" North DOS or CP/M, 8" CP/M), and a CRT console (preferably 80 columns by 24 lines or better). Use of a printer-terminal as the console is not recommended. Additional equipment to further enhance its capabilities includes additional disk drives (dual-sided access or hard-disk capacity useful), a North Star Floating Point Processor (highly recommended), up to 64K memory, a high-speed printer and a letter-quality printer.

-- 1.1 RUNNING PROGRAMS FROM THE OPERATING SYSTEM LEVEL --

APCBASIC operates under several different operating systems, which are programs that provide access to the various system resources such as the disk file system, printers, keyboards, CRT consoles, etc. Depending on which version of APCBASIC you have, a slightly different form of command must be typed from the console keyboard to execute a APCBASIC program. Under the North Star DOS, type the following command:

GO APCBASIC program

where ´program´ is the name of the file which holds the desired APCBASIC program. This command causes the DOS (disk operating system) to load APCBASIC which in turn loads your program file and then begins its execution. At that point your program takes over the computer and proceeds with whatever it is programmed to do. The program file contents must have been created by APCBASIC, North Star BASIC or another compatible BASIC.

Under the CP/M operating system the command sequence to run APCBASIC programs is quite similar:

APCBASIC program

The ´GO´ is unnecessary under CP/M but the result is the same. The program file is type .ZBA under CP/M and its contents should have been created by APCBASIC.

A 2nd form of APCBASIC is provided with the standard release for the production environment, as it furnishes more memory (about 3600 bytes more) to the running program as well as executing it about 50% faster than under the normal (development) APCBASIC version. This version is called simply: RUN. To use it instead of APCBASIC, substitute the name ´RUN´ for the name ´APCBASIC´ in the above command forms. RUN executes programs exactly as APCBASIC does in all respects except for its optimizing properties.

Remember that RUN and APCBASIC are simply files under your particular operating system and their names have no special meaning other than to identify the files to the system. (They must however be type1 files under North Star DOS and type.COM files under CP/M.) Their names may be changed or accessed from other drives, so read your system manual for further details on files and their naming conventions. Also see Section 7.4 for more on the CP/M versions of APCBASIC.

-- 1.2 PROGRAM DEVELOPMENT

To use the APCBASIC program development environment, type the GO APC[illegible] command described earlier, but omit the program file name. Without a program name, you immediately enter into the command mode of APCBASIC which under your direction provides facilities to create and debug programs. Only APCBASIC (the development version) provides this command mode, while RUN (the runtime production version) does not.

The command mode provides a selection of about 18 commands, from which you choose and enter from the keyboard. Each command performs a single task which APCBASIC performs after accepting the command. In this manner, commands may be entered one by one until your total task is completed. These commands are organized within Section 2.0 in the following way:

2.1 Program Entry & Retrieval	Entering programs from the keyboard or from files, listing your programs on the console or other devices, saving your programs to files.
2.2 Editing & Alteration	Sequential line editing, global search and replace, line renumbering, line range deletion, rearrangement of program sections, merging program modules from files into your current program.
2.3 Execution Control & Debugging	Running and testing, breakpoint and single-step debugging, interruption and continuation, interactive examination and setting of program data structures.
2.4 Miscellaneous Commands	Displaying program statistics, listing file directories, and exiting back to the operating system command level.

After entering the APCBASIC command mode, the first thing you do is either key in a program from the console or load an existing program from a file. To type new program lines from the console, enter a line number (an integer from 0 to 65535), followed by a sequence of program statements separated by semi-colons and terminated with a carriage return. Lines may be up to 159 characters long. The line number tells APCBASIC where to insert the new line into the current program. Therefore new lines may be entered in any order, providing a simple way to insert changes at a later time. See Section 1.3 below for further details on APCBASIC program format.

After entering or loading a program and making any desired changes, you can then run the resulting program under interactive control of execution to check its correctness. If errors are found, you can alter the program to correct the errors, and then repeat the process until you are satisfied with program operation. At any stage of the development phase, the current program may be saved on a disk file to safeguard your work from system failures or your own blunders (eg. power failures, mistaken revisions), or so that you may continue work at a later time. On completion of your working program, save the final version on a file for execution as described in Section 1.1.

-- 1.3 LINES, STATEMENTS AND PROGRAM FORM --

APCBASIC programs consist of a series of typed lines beginning with a line number and ending with a carriage return. Line numbers must be in the range 0 - 65535 and serve a dual purpose. First, since APCBASIC continually keeps the program lines arranged in ascending order, you can easily insert additional lines by typing them with appropriate line numbers. Secondly, some APCBASIC statements refer to program steps by line number, perhaps to repeatedly execute some group of statements or skip over undesired statements.

Besides being numbered, the lines themselves may be up to 159 characters long (two full CRT screen lines) and consist of one or more statements. Statements are separated from one another in the line with semi-colons (;) or backslashes (\) and exist as the fundamental process building-blocks of APCBASIC. Statements in general begin with a specific keyword followed by additional data parameters. By themselves, statements perform simple and easily understood operations, but in combination they can express procedures of unimaginable complexity. See Section 5.0 for a summary of the statement groups as well as full descriptions of every APCBASIC statement.

-- 1.4 INTRODUCTION TO THE APCBAS

A number of editing features apply whenever you are enter... ...or new program lines. These functions are invoked with special characters called control characters that are typed by pressing a specific character while holding down the key labeled CTRL on the left of the keyboard (the SHIFT key may be up or down). Not all keys perform editing functions and if accidentally struck will be rejected by the computer with a warning 'beep' sounded. For the purpose of notation 'CTRL-?' will denote a control character where ? is any character.

To delete a character mistakenly typed, follow it with the BACKSPACE key. Repeated use will remove successive characters one by one all the way back to the beginning of the line if necessary. This may be used to correct any input as often as necessary prior to terminating the line with the return key.

To delete the entire current line, (prior to terminating it with the return key), use CTRL-N. This is preferable to repeating backspaces as it eliminates unnecessary key strokes. Its effect may be cancelled if it is followed immediately by a CTRL-G, (see 1.6). CTRL-N will display a caret (^) followed by a carriage return to indicate it was typed, and allows you to re-enter your response to the last request.

-- 1.5 ADVANCED EDITING FEATURES --

Whenever you are entering data you may use the previously accepted response (the 'old line') in formulating the current entry (the 'new line'). This saves time when the computer requests successive entries that are identical or differ only slightly. The old line is always retained in memory for use in creating the new line. By using the control characters described in the following Section 1.6 you may edit the data in the old line and enter it as part of the new line as needed.

Each time you type a character into the new line it replaces the character available from the old line in the corresponding position. Backspacing, explained in the previous Section 1.4, retreats to the previous character in both the old line and the new line. By using the CTRL-A control character you may copy characters one by one from the old line to the new line. In this case the position in the old line is advanced to the next character. You may also suspend advancement in the old line to insert characters (using CTRL-Y).

Using these control characters is easy but it requires practice for them to be understood and become efficient tools. Each computer request for input offers an opportunity for unlimited practice. Enter a practice line and erase it with the CTRL-N before entering a carriage return. This makes it the old line. Now you may try out each of the control characters described in the following Section 1.6 to formulate a new line. As long as the return key is never pressed, this practice session may continue indefinitely.

-- 1.6 EDITING CONTROL CHARACTERS --

All editing control characters used by APCBASIC are described below. For convenience several alternates are provided which, for some, may correspond to function characters used in other systems for a similar purpose. These may be used interchangeably to your taste.

CTRL-A — Copies the next character from the current position in the old line into the new line. Repeat for successive characters.

BACKSPACE
DEL — Erases the last character typed in the new line and restores the corresponding character from the old line. Can be repeated to backspace all the way back to the beginning of the line.

CTRL-Z — Deletes the next character in the old line; a '%' is printed merely to indicate that a character was deleted. This control is used to skip undesired characters in the old line.

CTRL-Y
CTRL-E — Allows insertion of characters. The first CTRL-Y prints a '<' to show entry into this mode. After typing the characters to be inserted, a second CTRL-Y ends this mode, echoing a '>'.

CTRL-G — Copies all remaining characters from the old line to the new. Does the same thing as repeated use of CTRL-A, but with one stroke.

CTRL-D
CTRL-S — Copies all characters from the old line up to, but not including, a specified character. Operating like CTRL-A, this control permits copying long strings of characters with one stroke. After typing the CTRL-D, enter the character desired. Upper and lower case letters are equivalent. If that character is not in the old line a warning 'beep' is sounded and the sequence must be repeated. Remember that this is a two-stroke sequence.

CTRL-X — Deletes all characters from the old line up to, but not including, a specified character. Remember that this is a two-stroke sequence, just like CTRL-D above.

CTRL-N — Terminates the line, then permits re-editing from the beginning of the line. A caret (^) is printed to indicate this control was used. All remaining characters from the old line are lost.

CTRL-B — Copies the rest of the old line, then editing continues from the beginning of the line. (Same as a CTRL-G,N sequence)

CTRL-F — Copies the rest of the old line, then editing continues from the end of the line. (Same as a CTRL-B,G sequence)

CTRL-R — Throws away the new line and restarts with the old line that existed after the last carriage return, CTRL-R, CTRL-N, CTRL-B, or CTRL-F.

RETURN — The carriage return terminates the line-edit and copies the new line into the old line buffer for use in the next keyboard entry. This ends the current data-entry item and signals the computer it may now process that input item. Whatever stage of editing you were in has now been terminated. If a RETURN is typed as the first character of a line, an empty line is given to the requesting process and the program continues.

Syntactic Notation

To facilitate your understanding of APCBASIC commands (and statements), the following notation shall be used to simplify their syntactic description. Each command (or statement) consists of a sequence of typed symbols. The symbols are of two varieties: letters, digits and punctuation typed exactly as they appear in the command description, and general 'objects' that refer to kinds of symbol sequences to be typed. For example, DEL <line#>,<line#> requires the actual keyword DEL to be followed by two line numbers separated by a comma. Neither the angle brackets <> nor the characters they contain are actually typed: the angle brackets <> are always used to enclose the 'object' descriptions for which you must substitute specific values.

Optional items will appear within square brackets []. These brackets are not typed in actual commands (or statements) because they exist only to indicate portions of the command which may be omitted. Thus RUN [<line#>] means that an optional program line number may follow the RUN keyword. All other characters specified in each command (or statement) description must be typed exactly as shown.

Command and Statement Form

Most APCBASIC commands (and statements) that require multiple arguments have the form: <keyword> <argument list>, where the keyword is the name of the command (or statement), and the argument list may consist of strings, numbers, other keywords, etc. You must separate the list elements with commas, but no comma separates the keyword from the argument list. You may insert any number of spaces (or line-feeds) within your program to make commands (and statements) more readable. APCBASIC ignores all such characters not enclosed within quotes.

Specifying I/O Devices

Another common feature in APCBASIC commands is the I/O device used for any data input required or output generated. Denoted below as #<device>, this is a value from 0 to 15 preceeded by a lb-sign (#) to signal its presence within the command. This is necessary since <device> is always an optional feature whose omission defaults to device #0, the console. Device #1 refers to the printer. See Section 5.2 for additional details about the devices.

Specifying Program Line Ranges

Several APCBASIC commands (LIST, DEL, REN, CHANGE) have the option to restrict their operation to a line range within the program. Specified as <line# range> in the discussions that follow, a line range consists of a starting and ending line number pair separated by a comma. A dollar sign ($) may be used to designate the last line of the program. For example, LIST $ or DEL 200,$.

Search Strings

Certain other commands (LIST, EDIT, CHANGE) restrict their operation to lines which contain a pre-determine character pattern called a search string. This string is specified in the command and used internally to select the lines for processing. A search string is formed by simply typing the characters you wish to match. If the search string contains commas (used as an argument separator in some commands), you must surround the string with quotes ("").

-- ALPHABETIC COMMAND SYNTAX SUMMARY --

CHANGE <line# range>,<search string>,<replacement string>
CONT
DEL [<line# range>]
DIR [#<device number>,][<drive number>]
DOS
EDIT [<starting line#> [, <search string>]]
ENTER #<file number>
ENTER [<starting line number> [,<stepsize>]]
LIST [#<device>,] [<starting line#>[, [<ending line#>[,<search string>]]]]
LOAD <program file name>
MERGE <program file name>
REN [<new start#> [, <stepsize> [, <old start#> [,<old end#>]]]]
REN$ <new start#> [,<old start#> [,<old end#>]]
RUN [<line#>]
SAVE <program file name> [<file size>]
SIZE
TRACE RET
TRACE [#<device>,] IF <logical exprn>
TRACE [#<device>,][<line#>]

Several of the above command keywords may be abbreviated to a specific two or three character sequence. These have been indicated above in bold face.

ENTER [<starting line number> [,<stepsize>]]

At any time you can type a line number followed by a line of statements and APCBASIC will insert it into the appropriate place in the program. The ENTER command provides automatic line numbers for a series of new lines that your enter. You may optionally specify a starting number and stepsize; if either is omitted it defaults to 10. To terminate the process, type either a CTRL-C or just a carriage return immediately after the automatic line number appears.

ENTER #<file number>

APCBASIC programs are normally stored in a special coded form on the file. Occasionally, you may have a text file containing program text from another system or different dialect of BASIC that you wish to convert to the APCBASIC system. Such a file may be entered into APCBASIC with this special form of ENTER. Do not use this to enter programs already stored by APCBASIC (or North Star BASIC) with the SAVE command. LOAD is provided for that purpose.

Before using this command, you must OPEN the text file under a file number ranging from 8 to 15. Having done that, simply say ENTER #N, where 'N' is the file number assigned. APCBASIC will input each line from the text file, code it into its internal form and display it on the console. This process continues to the end of the file according the following rules:

(a) Each line is terminated with a carriage return (ASCII 13).

(b) When a line-feed and a carriage return appear in pairs, the 2nd character of the pair is ignored.

(c) Empty lines, consisting of only a carriage return, are ignored.

(d) A control character (ASCII codes 0 - 31) or any character with bit 7 on (ASCII codes 128 - 255) as the 1st character of a line signals that the end-of-file has been reached.

(e) Control characters elsewhere in the line will appear as question marks (?) and will have to be removed in a later editing phase.

(f) Each and every line must begin with a line number in the range 0 to 65535. Otherwise a LINE# ERROR is issued and the input process terminates. APCBASIC inserts each line into its proper sequence according to its line number, so they need not be in order.

(g) Lines are limited to a maximum of 159 characters. Additional characters beyond this limit are ignored and lost.

(h) The input process may be interrupted by typing a CTRL-C. To resume input after a CTRL-C or a LINE# ERROR, just retype the original ENTER #F command (possibly with a CTRL-G and a return).

LIST [#<device>,] [<starting line#>[, [<ending line#>[,<search string>]]]]

Provides a display or printout of your program on the device specified, or on the console by default. With the exception of the <device> which is purely optional, all remaining arguments are optional but may only be omitted from right to left, as indicated above. For example omitting the <ending line#> means that no <search string> may be supplied either. With both line numbers omitted the entire program is LISTed. With only the start number given, just that line is displayed. To LIST through to the last program line use a dollar sign ($) to denote the last line as shorthand for the its line number.

APCBASIC uses the optional string argument to search through the line range given and list only those lines containing that string. Question marks (?) may be included in the string to act as 'wild card' characters that match any character (this feature is described further under the EDIT command in Section 2.2). Upper and lower case letters are seen as equivalent.

Since LIST displays your program as fast as your output device will accept characters, a APCBASIC provides a simple method to control this speed to suit your requirements. During a LIST, you can type the space-bar to start and stop the display. Once stopped, you can type a carriage return (or line-feed) to display successive program lines one at a time.

The <device> number need only be specified to direct the program listing to an output device other than the console (device #0). Usually this would be the printer (device #1), but may also be a file (devices #8 through #15). The resulting file contains pure text and is unsuitable for subsequent LOADing, but it can be processed by other text file utilities (eg. text editors or formatters) which cannot handle the coded format of 'normal' APCBASIC program files. See Section 5.2 for further details about text file processing.

LOAD <program file name>

Erases the current program, if any, and then LOADs another program from the file specified. The program must have previously been created and saved by APCBASIC (or North Star BASIC) to be valid. Errors in this process occur from LOADing an improper program or from insufficent memory to hold the desired program. After such an error, APCBASIC will maintain as much of the program as it could so that you can determine the point of error.

SAVE <program file name> [<file size>]

SAVE will save a program on a new or existing file. In either case APCBASIC will respond with an OLD or NEW FILE message and request user confirmation of the SAVE (Y/N response). A file length specification may follow a new file name to assign the initial file capacity (under North Star DOS only). Omitting this size for a new file defaults to a file size that is 3 file blocks larger than the program SAVEd.

If you type SAVE without a file name, APCBASIC uses the file name specified in the previous successful SAVE command. This feature makes program backup during development fast, repeatable and mistake proof. If no prior SAVE has been issued since a start-up or a prior LOAD command, SAVE without a file name results in an ARGUMENT ERROR.

EDIT [<starting line#> [, <search string>]]

Provides the opportunity to edit your program lines sequentially from a specified starting line number, or from the beginning of the program if not specified. Each line is displayed, then placed in the edit-buffer for unlimited editing. Only on a carriage return will APCBASIC accept this line and proceed to the next one in the program. You may skip over a line (leaving it unchanged) by typing an immediate carriage return, instead of performing some editing function. See Section 1.5 for all the details on the editing control characters.

Since editing always continues with the first line after the current line entered, you can re-start the editing sequence anywhere in the program by simply typing an unused line number at the desired starting point (followed by a carriage return). This normally deletes the line there, so the number you pick must be unused. The EDITting mode persists until you enter a line without a line number (such as a command or direct statement) or you type a CTRL-C or the end of the program is reached.

By including a string parameter, APCBASIC will provide editing to only those lines containing that string. For example the command: 'EDIT 100,FILE' displays each line after line# 100 that contains FILE, allowing you to edit them. Upper and lower case letters in the search string are equivalent. You must enclose the string with quotes ("") if it contains any commas, for example: EDIT 100,"A$(I,J)"

For flexibility, your search string may contain special 'wild card' characters that match any character. This special character, a question mark (?), may appear anywhere in the search string (except as the first character) and as many times as desired. With this concept, the string A??= will match all assignment statements with variable identifiers 3 characters long beginning with the letter 'A'.

CHANGE <line# range>,<search string>,<replacement string>

Global search-and-replace may be done with the CHANGE command, which replaces one string with another everywhere or selectively within a line range. The use of the line range and the <search string> is identical to the LIST command, except that all four arguments are mandatory. APCBASIC will request 'VERIFY?' to allow user control of each replacement ('Y' response). An 'N' response causes automatic replacement of all occurences found. In either case APCBASIC displays the number of replacements made on completion.

The CHANGE command does not permit access to the line number portion of program lines. Be sure to use the VERIFY option after you specify a number search in a CHANGE command, as it may match portions of line numbers in GOTOs, GOSUBs, etc. Changing line numbers should only be attempted with the REN command.

To permit commas (,) and lower case characters in string arguments of the CHANGE command, the string argument must be surrounded by quotes (""). For example: CHANGE 1,$,"A(I,J)",Z(R) will change A(I,J) to Z(R). Such quotes may not be used as ordinary characters in either string.

DEL [<line# range>]

Deletes the specified line range from your program. If no range is supplied then APCBASIC erases the entire program, but first requests verification to prevent unintentional erasure. Note that a dollar sign ($) denotes the last line of the program. See Section 5.7 for a discussion on executing DEL from APCBASIC programs. Single line deletions are best performed by typing the line number followed immediately with a carriage return.

REN [<new start#> [, <stepsize> [, <old start#> [,<old end#>]]]]

Provides a general program renumbering facility that renumbers any range to any other range. APCBASIC does not permit any renumbering that would cause line interleaving or duplicate line numbers. However it does support rearrangement of whole groups of lines as well as 'simple' renumbering, given appropriate instructions.

All arguments are optional, but must be omitted from right to left (eg. the start# and stepsize must be present in order to specify a line range). When left out, the following defaults are used: <start#>=10, <stepsize>=10, <old start#>=1st program line, <old end#>=last program line. By omitting the <old end#> from the line range, the range RENumbered includes of all lines from the <old start#> to the end of the program. APCBASIC adjusts all references made to lines renumbered by the process, however unsatisfied references remain unchanged. The following examples illustrate how REN may be applied:

Example REN Command	Effect on the Program Lines
REN	Renumbers the entire program from 10 by 10s.
REN 250	Renumbers the entire program from 250 by 10s.
REN 375,5	Renumbers the entire program from 375 by 5s.
REN 500,12,2000	Moves lines numbered 2000 and up into the range 500, incrementing by 12s.
REN 200,3,800,899	Moves all lines in the 800 range to 200 by 3s.

APCBASIC always validates the implied operation that you request and aborts with an OUT OF BOUNDS ERROR to prevent interleaving of lines or illegal line numbers. References throughout the program to altered line numbers are updated properly. Line number references to non-existent lines will remain unchanged.

REN$ <new start#> [,<old start#> [,<old end#>]]

When you wish to RENumber program sections without disturbing the current line increments, a different REN form is available. It works exactly like the above form except that no <stepsize> may be specified. When omitting the <old end#> line number, the last program line# is assumed. When omitting any specified old line range, the entire program is RENumbered. The <new start#> parameter MUST be supplied, which specifies the new starting number for the range given.

MERGE <program file name>

The MERGE command provides a generalized facility for adding APCBASIC code lines from other files to your currently LOADed program. Line ranges in both current program and source file are unrestricted. On duplicate line number conflicts, MERGE replaces the old line with the new line from the file, while the remaining lines are inserted into their proper places. Meaningless code may result from interleaving lines indiscriminately.

MERGING large files with large programs can sometimes be quite slow, up to a minute in rare cases. Since APCBASIC loads the entire file before merging its lines, you must have sufficient memory for this operation to succeed. See Section 5.7 for a description on the executable MERGE.

RUN [<line#>]

Causes program execution to begin at the first program line or starting from a line specified. If the program has been interrupted (see CTRL-C below), a RUN command erases all previous data before beginning execution. In most cases the optional line number will be never be specified, but this can be useful when several independent programs reside within the program text (usually for debugging purposes).

CTRL-C

This is not a command, but a control key used for stopping whatever process is currently underway: sort of a panic button. When CTRL-C is struck during execution of a program, it performs exactly the same action as a programmed STOP. This is useful during the debugging phase to see where execution is currently happening or to immediately terminate an erroneous program.

CTRL-C stop may be trapped by the ERRSET (in Section 5.4) statement as a type 15 error, allowing program control over the computer response to a CTRL-C. Also, the PARAM(1) statement (in Section 5.8) can enable/disable the CTRL-C apparatus during program execution. Disabling CTRL-C permits control over user intervention and speeds up execution within tight loops by a small but noticable amount.

When CTRL-C is typed in the command mode, the current entry or command is aborted but no STOP message will be issued, only READY. The STOP message is given only to indicate the interruption of a running program. The CTRL-C break character is recognized only from the console device and is just another control character when input from devices 1 to 15.

CONT

Resumes program execution after a CTRL-C or programmed STOP. The prior state of program execution is restored (DATA READ pointer, ERRSET status and program location) before CONTinuation. You cannot CONTinue after changing the program in any way (eg. by EDIT, CHANGE, ENTER, etc). Between the STOP and subsequent CONTinue, you may execute direct statements (below) without losing the ability to CONTinue. APCBASIC preserves all variables and OPEN files, but they may be altered by direct statements while in the command mode.

DIRECT EXECUTION

Any executable program statement or line of statements may be typed as a command (ie. without a line number) causing immediate execution of what you typed. For instance you can interrupt a running program and then display the contents of an entire array before continuing. Or you might want to use APCBASIC as an intelligent calculator by displaying complex numerical expression values. Direct execution is a very important tool for debugging programs, but may also be employed for learning the properties of any APCBASIC statement.

Direct FOR..NEXT loops execute properly only when the entire loop is entered as one line (159 characters maximum). Direct expressions may access currently DEFined User-Functions. But since these functions only become defined after execution begins, you must interrupt (CTRL-C) the program or it must STOP or END before such access becomes possible. GOSUBs may be executed without any difficulty, but GOTOs cause a CONTinuation (see above) at the line number specified. A direct RETURN also CONTinues program execution, but the program must have been interrupted within a GOSUB. You can alter the contents of program variables (eg. X=FNT(Y)+SQRT(Z)), alterations which carry over to CONTinued execution.

TRACE [#<device>,][<line#>]

The TRACE command provides an excellent environment for debugging APCBASIC programs and may be invoked prior to RUNning or CONTinuing your program. TRACE without arguments breaks subsequent program execution to display the remaining unexecuted statements of the current line and suspend further program execution until a TRACE control character is typed. In the break mode, the following control characters are recognized:

SPACE BAR — Executes the next statement and breaks (single-step mode). By hitting the space-bar repeatedly you can watch program execution one statement at a time.

ESC — Terminates the TRACE mode and continues program execution. This is the only way to terminate the TRACE mode. The only way to reinstate the TRACE mode is to interrupt your program with a CTRL-C, enter another TRACE command, then CONTinue program execution.

CTRL-R — Resumes normal program execution (without displaying program statements) until the current line is again be processed.

CTRL-N — Resumes program execution until execution passes to the line following the current line.

^ — Resumes program execution until the current sub-program or loop terminates (GOSUB, FN, FOR-NEXT, WHILE-NEXT). This permits you to ignore the remaining details of any loops or sub-programs you happen to fall into while tracing your program.

CTRL-T — Resumes program execution until a line number transfer occurs, such as a GOTO, GOSUB, ERRSET trap, etc. This permits you to skip sequences of in-line programming which are not of detailed interest at the time.

CTRL-C — A CTRL-C will enter the command mode (causes a STOP to be executed), so that commands & other direct statements may be entered, and the TRACE mode can be resumed by using CONTinue or terminated by entering TRACE without a line number. ERRSET (in Section 5.4) will not trap a CTRL-C with TRACE in effect.

APCBASIC rings the bell (sends CTRL-G to console) to indicate an illegal control character was typed in the break mode. For debugging convenience, if a LINK to program occurs while the TRACE mode is active, no automatic size reduction (space & REM removal) will take place. If during the TRACE mode an automatic LINK, MERGE or DELETE (from Section 5.7) is encountered, the execution breaks at the first statement of the program (ie. breaks on completion of segmentation or overlay statements). Once invoked, the TRACE mode persists until terminated with the ESC control or an untrapped error occurs during program execution.

Typing TRACE followed by a line number and then RUNning the program causes execution to break on reaching that line. The line is displayed and execution is suspended until a character is typed on the console. You can direct TRACE output to a device other than the console by specifying the device number immediately after the TRACE keyword. However all TRACE control characters are always accepted from the console keyboard (device 0).

TRACE [#<device>,] IF <logical exprn>

To conditionally TRACE your program, use the TRACE IF <expression> command, where the expression may be any logical or numeric expression. When running, the program proceeds as usual until the expression becomes TRUE (non-zero). The TRACE then begins and continues until the expression becomes FALSE (zero), during which time you may use any of the controls described above. Note that complex expressions will slow execution considerably. Any errors in evaluating the expression will be reflected as errors in the line being executed. All TRACE control characters are available during the break to further control or modify the TRACE operation. By using the conditional TRACE you can determine dynamically how and where erroneous values originate.

TRACE RET

This command will display the RETURN path (in line numbers) after the program stops for any reason (eg. CTRL-C, STOP, program error, etc). If the program is CONTinuable, the first line number displayed will be the point of CONTinuation. The RETURN path is displayed all the way back to the first FN or GOSUB reference that began the sequence. Since this may be quite long, you can abort this display by typing a CTRL-C (GOSUBs and FNs can descend hundreds of levels in APCBASIC). TRACE RET operates only in the command mode and has no effect on the dynamic TRACE mode if set.

SIZE

Displays your current program size in number of file blocks (units of 256 bytes). APCBASIC also prints the program size automatically after a program SAVE, LOAD, or MERGE (see Sections 2.1 and 2.2).

DOS

Causes an immediate return to the operating system level. Control may be regained by causing a JMP to memory location START+14H without disturbing the state of the program, possibly in progress, that you left.

DIR [#<device number>,][<drive number>]

Prints a copy of the file directory from <drive> to <device>. When omitted, both arguments take on the current default assigned to them (see Section 5.8 for the PARAM() statement). Normally the defaults for the <device> and <drive> are 1 and 0 (console) respectively. DIR is also an executable statement (see Section 5.6), so that your APCBASIC programs can also list file directories.

**** 3.0 REPRESENTING AND MANIPULATING NUMBERS ****

APCBASIC possesses two fundamentally different data representations: Numbers and Strings. The numeric representation supports BCD floating point in fixed precision in the range -10^63 and 10^63 (exclusive). This section describes in depth the concepts and use of numeric constants, variables, arrays, expressions and functions. Strings and their associated representations and operations are discussed in Section 4.0.

-- 3.1 NUMERIC CONSTANTS --

The most obvious numeric form is the constant, which is a number in the usual sense of the word. Examples are: -1, 5675261, 4.536, 0, -11.111, 00934.2, etc. Constants may be signed or unsigned, but numbers like 1,435 are not permitted, since commas imply separate objects. The smallest numeric value permitted in APCBASIC is 10^-64. Arithmetic operations producing smaller numbers than this always result in zero (ie. underflow => zero).

Constants may be expressed in so-called E-notation as well. Similar to scientific notation, this format includes a scaling factor to indicate a power of ten multiplier. For example, 23.4104E-2 and .234104 are identical values with the first in E-notation. This representation becomes important when extremely large or small constants are required. For example the constants: -.20152E42 and 3.3142E-19 would not be humanly digestable with all the zeros needed to represent them in standard notation.

Constants are used within programs to represent fixed quantities which are needed in computations when forming results. Constants may also be entered from the keyboard in response to requests from the computer as directed by the program.

-- 3.2 NUMERIC VARIABLES --

As in most other programming languages, numeric variables in APCBASIC provide the means for storing numbers for later access. Variables represent numbers just as constants do but with one big difference: the quantity they represent can be altered by storing a different value into the variable. See Section 5.3 for details on storing values into variables with assignment statements.

Numeric variables are identified in your programs by a one or two character name. The first character must be a capital letter (A-Z) and the optional 2nd character must be a digit (0-9). For example X, X2, A8, Z1 and Y are legal variable names, while XX, R0, TOT and 1Z are not. Variable names may be used wherever numeric values are expected, causing access to their current stored value. Numeric variables may be later referred to as scalar variables, simple variables, or just variables to distinguish them from array variables described later in Section 3.3.

APCBASIC numeric variables as in all programming languages maintain only a finite amount of precision. In the standard version, 8 decimal digits is the maximum precision possible. Constants are rounded to the nearest 8th digit whenever 9 or more digits are specified. Precisions of 10, 12 and 14 digits are similarly rounded. The number of memory bytes assigned to each variable value depends on the prevailing precision. If P represents the number of digits precision, then the number of bytes required is given by the expression: 1+P/2.

-- 3.3 NUMERIC ARRAYS --

Another type of numeric variable is the array, in which a group of numeric values can be stored under one name. Arrays are organized as an ordered set of storage locations, called elements, that are identified by a position number within the ordering. For example A(0), A(1) and A(2) represent the first three elements of array A(). Parentheses are used to indicate that A() is an array and serve to contain the position of the desired element. The positions are numbered from zero by integers up to the size of the array. Since APCBASIC can distinguish between variables and arrays of the same name (eg. A(3) and A), such variables can co-exist in the same program without conflict.

The arrays described above can be conceived of as a column of numbers with positions numbered from zero down the side. Suppose that we have many columns side by side and that we number them from zero along the top. This is called a 2-dimensional array. By identifying the row and the column we can locate any element of the group. For example A(I,J) refers to the element of A() in Row(I) of Column(J), where I,J are simple variables containing the element position. By adding further levels to this idea, 3 or higher dimension arrays can exist. An N-dimensional array requires N position numbers, called subscripts, to uniquely specify an element in the array.

DIMensioning Numeric Arrays

In order for an array to exist it must be defined in your program prior to its use. The information necessary consists of the array variable name and the range of valid positions for each dimension, which you can provide in the DIM statement. DIM A(50),B(12,15) defines array A() as a 1-dimensional array with element positions 0-50, and array B() as a 2-dimensional array with row positions 0-12 and column positions 0-15. Dimension postions always begin at zero and continues up to and including the limit specified for that particular dimension. An OUT OF BOUNDS ERROR occurs if you attempt to access a dimension position outside its range. One DIM statement can define one or more arrays by simply listing their definitions one after another separated by commas. All array elements are initialized to zero when DIMensioned. If you refer to an array without DIMensioning it, APCBASIC implicitly DIMensions it as a 1-dimension array with positions 0-10.

Once an array has been DIMensioned, all references to it must specify positions for each dimension defined. For example an error results from the reference B(3) to the 2-dimensional array example above. However you can re-DIMension the array at any time (except within FNs and FOR loops) by re-defining it in another dimension statement. All stored values are erased after such an operation and set to zero. In this manner arrays can grow or shrink depending on your program requirements. When arrays are made smaller the unused memory space is available to the system for use elsewhere. Never DIMension arrays inside FOR..NEXT loops (Section 5.4) or inside User-DEFined Functions.

— 3.4 NUMERIC EXPRESSIONS —

The fundamental computational structure in APCBASIC is the expression, which is constructed from data symbols and operation symbols, much like algebraic notation. Expressions permit you to specify a number as a combination of other numbers. For example (2+5)*3 represents 21 by arithemetically combining 2, 3 and 5. In general, you may use expressions wherever numbers are needed.

Data symbols can be constants (representing fixed quantities), variables (storing the data used), functions (returning computation results), or sub-expressions. A sub-expression is actually another expression enclosed inside parentheses to group it as a computational unit.

Operation symbols, called operators, are of two types: unary and binary. Unary operators act on a single number to form a single result number. For example the unary minus operator (-) causes negation of a value that follows it (eg. -X). Binary operators however act on two numbers to form one result. For example the binary plus operator (+) forms the sum of two values (eg. X+5).

APCBASIC evaluates expressions by proceeding left to right, accumulating the result with each operation as it goes. The various operators are not however applied with equal priority. Take the following expression using addition (+) and multiplication (*) for example:

```
2 * 3 + 7 * 8
```

If we apply the usual rules of algebraic evaluation to this expression, we would perform the multiplications prior to the additions (in the absence of parentheses). Thus multiplication is said to take precedence over addition. Similarly, a priority scale has been assigned to all operators in APCBASIC which can modify the order in which their operations are performed, according to the generally accepted rules of algebraic evaluation. You can override the default operator precedence as needed by surrounding a sub-expression with parentheses (sub-expressions take precedence over all operators). To evaluate the addition in the example before the multiplications, just write it like this:

```
2 * (3 + 7) * 8
```

All numeric operators are listed below in order of decreasing precedence and followed by a brief description of each.

Numeric Operators in Decreasing Order of Precedence

(0) Evaluate Constants, Variables, Functions and Subexpressions
(1) NOT and Unary Minus (-)
(2) Exponentiation (^)
(3) Multiplication (*) and Division (/)
(4) Addition (+) and Subtraction (-)
(5) Comparision Operators: =, <>, <, >, <=, >=
(6) AND
(7) OR
(8) XOR (Exclusive OR)
(9) EQV (Equivalence)

Operators on the same line have equal precedence and are evaluated from left to right as encountered.

ARITHMETIC OPERATORS

The arithmetic operators are the most familiar and simplest to describe. The left and right operands around an arithmetic operator are simply combined algebraically into a result value using the specified operation. APCBASIC includes the following five operators:

OPERATOR	NAME	D E S C R I P T I O N
+	ADD	Forms the algebraic sum of the two operands.
-	SUBTRACT	Algebraically subtracts the right operand from the left operand.
*	MULTIPLY	Forms the product of the two operands.
/	DIVIDE	Divides the left operand by the right operand.
^	POWER	Raises the left operand to the power specified by the right operand. X^0 is always 1, even if X=0. OUT OF BOUNDS ERROR if X is negative and the power is a non-integer or outside the range [-9999,+9999], or if the attempted result is greater than 10E62.

LOGICAL OPERATORS

Logical operators are unusual in that they do not use the numeric value of their operands. Instead, they only look at their operands as being zero or non-zero, ie. values are of two classes: zero and non-zero. Think of this property in terms of TRUE and FALSE, with TRUE being non-zero and FALSE being zero.

The result of a logical operation is always zero (0) or one (1) and reflects the combination of two logical values into one logical result. NOT causes a logical reversal of the logical value following it, ie. NOT FALSE = TRUE = 1, NOT TRUE = FALSE = 0. See the truth-table below for complete definition of the logical binary operators.

TRUTH TABLE DEFINITION OF LOGICAL OPERATORS

LOGICAL OPERATION	LEFT ARGUMENT	RIGHT ARGUMENT	LOGICAL RESULT
NOT	--	False	True
	--	True	False
AND	False	False	False
	True	False	False
	False	True	False
	True	True	True
OR	False	False	False
	True	False	True
	False	True	True
	True	True	True
XOR	False	False	False
	True	False	True
	False	True	True
	True	True	False
EQV	False	False	True
	True	False	False
	False	True	False
	True	True	True

Another way to look at these logical operators is as follows. AND results in TRUE only when both operands are also TRUE. OR results in TRUE only if at least one of its operands is TRUE. XOR (exclusive OR) results in TRUE if one operand is TRUE and the other is FALSE. Finally EQV (equivalence) results in TRUE only if both operands are TRUE or both are FALSE.

COMPARISON OPERATORS

Comparison operators are similar to logical operators in that their result is the logical value 0 or 1. However their operands can be either numeric values or string values. Each comparison operator compares its operands and returns TRUE or FALSE (1 or 0) indicating the outcome of the comparison. Both operands must be of the same data type (strings with strings, numbers with numbers).

OPERATOR	DESCRIPTION
=	Equality
<	Less-Than
>	Greater-Than
<=	Less-Than or Equal
>=	Greater-Than or Equal
<>	Not Equal

See Section 4.4 for the discussion on string comparision operations.

— 3.5 APCBASIC FUNCTIONS —

Data can be represented in expressions not only as constants and variables but also as results of special procedures called functions. For example SQRT(17) is a built-in function that represents the square-root of the constant 17. Functions are always of the same form:

<function name> (<argument list>)

In the above example the argument list consists of one element: 17. Enclosed in parentheses, the argument list contains data which the function uses to produce a single string or numeric result. Arguments may be string or numeric expressions whose number and type depends on the particular function being used. APCBASIC possesses a library of about fifty built-in functions and provides facilities for user-defined single and multiple line functions.

Built-in Function Library

Section 6.0 provides a complete description of all the built-in functions in APCBASIC. These should be studied for utility in your particular applications since there may be several that already do what you have in mind and they run many times (even hundreds of times) faster than similar procedures written out in APCBASIC. They are grouped into arithmetic functions, mathematical functions, string functions (including bit operations), file & device I/O functions, and functions for system interface. This function set has been carefully selected to cover the widest variety of applications with a minimal number of separate function entities (when applied in combination).

The remainder of this section is devoted to the definition and application of user-defined functions. Refer to Section 4.0 for details on strings if you encounter difficulty with any string concepts used below.

User-DEFined Functions

User-defined functions are identical to built-in functions except the procedure that uses the argument list to produce a result is constructed from APCBASIC statements. In this manner you can define a procedure once and refer to it as often as necessary anywhere else in your program without having to duplicate it each time, saving programming time and memory space in the process. The arguments are passed to the function procedure as parameter variables which the procedure uses for input data. The data type (string or numeric) of the parameter variable defines the argument type to be employed. This information is entered into the DEF statement portion of a function definition, described later.

As with library functions, user-defined functions are named and include an argument list. The name consists of the two identifying characters 'FN' followed by an ordinary variable name (eg. FNZ, FNX0, FNA$, FNR3$, etc). A dollar sign ($) ending the name indicates that the function generates a string result; functions named without a dollar sign must generate a numeric result.

The argument list is enclosed in parentheses and may contain string or numeric argument expressions, but both their number and type must correspond to your definition of the function (described below). The argument list need not exist if no arguments are required for its operation. For example FNT$ might be such a function that returns a string containing the current time of day. Of course FNT$ must be defined without a parameter list.

Single Line User-DEFined Functions

User-function definitions come in two varieties: single line and multiple line functions. Single line functions must be contained within one line (159 characters maximum) and take the following form:

DEF <function name>[(<parameter list>)]=<expression>

where <function name> specifies a unique function name reflecting the data type of the result, the optional <parameter list> lists in parentheses the variables through which the argument data is passed to the function body, and <expression> specifies a string or numeric expression combining the parameters (with possibly other data) into a new result of the data type specified by the function name.

This form of function definition provides a simple way to combine data using a complex expression, particularly when that expression is required in many places of the program. For example: DEF FNM(N,M)=N-INT(N/M)*M. The variables N and M are the function's parameters and will be used to represent the data presented by an actual reference to the function: FNM(X-17,SQRT(Y)). Numeric parameter definition variables have no relation to variables of the same name used outside the function definition. This type of variable is called a local variable because the scope of its existence is localized to a subset of the program. String parameter variables and all other variables may be accessed throughout the program and hence are called global variables.

Remember that each call creates new local variables (numeric only) which receive the arguments. After their data has been used, local variables can serve as temporary storage that lasts only as long as the current call. If more local variables are desired you can use a LOCAL statement (in Section 5.5) to create them. This method also provides LOCAL string variables when desired.

String arguments may be passed to user-defined functions but the local variable approach is not employed. Instead the actual string argument is copied to the corresponding string variable of the definition parameter list. Data present in the parameter variable is overwritten and lost. In addition, the size of a string argument is limited to the DIMensioned size of its corresponding parameter string variable.

Multiple Line User-DEFined Functions

Multiple line user-defined function permit construction of functions with any number of statements. The definition has three parts: all DEF statement similar to the above, the main body of the function procedure, and a FNEND statement to terminate the definition. Use the following form of the DEF statement for multiple line functions:

DEF <function name>[(<argument list>)]

The only difference between this and the single-line DEF statement above is absence of the equal sign (=) and <expression>. Instead, the main body of the function procedure immediately follows the DEF statement. This main body is an unrestricted APCBASIC procedure (except that it cannot include another DEF statement) that performs the desired task. However it must do something special to pass the result back to the expression that used the function. To do this, the RETURN <result exprn> is executed which computes and transmits a result back, then causes program excution to continue from the point it left off when the function was used. The <result exprn> must compute a numeric result for numeric functions and a string result for string functions. For example:

```
100 DEF FNN0(N$); ERRSET 120,E,E
110 E=VAL(N$); RETURN 1
120 RETURN 0; FNEND
```

This simple but useful function logically tests a string for whether or not it correctly represents a numeric constant. It returns 0 (false) if an error occurs when VAL(N$) attempts to convert N$ to a number, or returns 1 (true) if successful. Thus FNN0(" 234.017 ")=1 and FNN0("1,047,471")=0.

Multiple line functions must have FNEND as their final statement. You cannot define other functions within function definitions, but you can define them in terms of other functions by employing user-defined functions as components in forming higher level results.

Recursive Programming

You are free to employ the function you are defining within its own definition. Known as recursion, such functions must utimately reduce down to a result without reference to itself in order to terminate in a finite amount of time. Otherwise they continue to invoke themselves until all the memory in the machine is consumed, ending in a MEMORY FULL ERROR. Recursive functions often split a problem into several smaller but similar problems, then call themselves to solve each of these.

Side-Effects caused by User-DEFined Functions

User functions permit execution of APCBASIC procedures within the evaluation of APCBASIC string and numeric expressions. When the function RETURNs, expression evaluation resumes where it left off and the program continues. Ideally, the context in which the function is used should be unaffected by the act of calling the function, other than the result generated. However there are two areas where function calls can potentially upset program integrity in non-trivial and unobvious ways, which are known as side-effects.

The first side-effect is the problem of global variables changed within the function procedure and is one of the most frequently encountered sources of programming errors when programming with User-DEFined Functions and GOSUBs. In the example below, variables outside the function are affected by the function call causing a FOR..NEXT loop to continue 'forever':

```
200 FOR I=100 TO 1 BY -1; A(I)=FNT(I); NEXT; END
800 DEF FNT(N); T=0
805 FOR I=1 TO N; T=T+B(I); NEXT
810 RETURN T; FNEND
```

You must ensure that this kind of situation never happens in your programs by restricting potentially harmful variable accesses. Two mthods are available for controlling variable access. Every time you use a variable, find out how and where it is used elsewhere in the program. Use the ZBIG utility program (Section 8.1) to print an index to your program structures and all references to them.

Data stored in variables must of course be preserved for the term of its usefulness. Variables which contain long-term data must be protected from unintential use, especially in large programs. You may safely obtain temporary storage by using available FN parameters (if any), or by temporarilary creating new variables with the LOCAL statement (Section 5.5). Both methods should be employed but use local variables wherever possible.

The 2nd type of side-effect is quite obscure, but you should be aware of it. READ or WRITE statements containing user function calls which in turn perform their own READs or WRITEs to the same file, can upset the current file position causing the original READ or WRITE to access the wrong file position. For example if you directly WRITE the result of a function that itself accesses the same file, the data will be written at the file position left by the function call rather than the position specified by the original WRITE statement. Or suppose that a READ statement includes a user function that CLOSEs the file in its procedure. Such an operation would produce highly unpredicable results. Awareness of this side-effect is essential to prevent it from occurring. You can always store the function result in a variable for subsequent use in READ or WRITE statements to avoid such difficulties.

**** 4.0 REPRESENTING AND MANIPULATING STRINGS ****

APCBASIC possesses two fundamentally different data representations: Numbers and Strings. Numbers and their associated operations are fully described in Section 3.0. Strings are series of adjacent characters (8-bit bytes) used to represent anything from text to integers to arbitrary binary information. Their representation and manipulation is fully discussed in this section.

-- 4.1 STRINGS AND STRING CONSTANTS --

Most typical business application programs spend most of their time dealing with strings: word processing, mailing lists, report generation, command processing, record processing, and formatting to name a few. Strings can represent binary information, text, packed numbers or virtually any other data representation. APCBASIC has a carefully chosen set of operations which when used in combination can efficiently perform all string operations supported by PL/1 or other high-level computer language with exceptional string handling facilities. Becoming fluent in APCBASIC string handling concepts can greatly simplify many of your non-numeric data processing applications.

A string is a sequence of bytes treated as a single data object. For example the string constant "This is a String" is a 16-byte string. The quotes are used to clearly separate the string characters from those around it but are not actually part of the string. Only double quotes (") may serve this purpose and single quotes (') may not, which precludes use of the quote character (") within a string constant. String constants are sometimes called literals. Of special note is the literal "", which is a string of zero bytes called a null string.

String constants are used in programs to represent fixed character sequences (usually text) which are manipulated with other strings to form string results. This is analogous to the use of numeric constants (Section 3.1) in programs as fixed quantities. Because string constants are fixed sequences of printable ASCII characters, their utility is limited and mostly found in PRINT statements (Section 5.2). A more general string representation called string variables will now be discussed.

-- 4.2 SIMPLE STRING VARIABLES --

Character strings may be stored in string variables for later retrieval by name. Their names are similar to numeric variables but always end with a dollar sign ($) to indicate a string variable name. For example the string variable names A$, Z$, R0$ and W2$ are legal, while X2, 5$, $A and TEXT are not. Using string varible names wherever string data is expected gives access to the data stored in the variable. Assigning string data to a string variable replaces its previous contents with the new string and can be performed by assignment statements, EXAM statements or (file or data) READ statements (all in Section 5.0).

Unlike string constants, the characters stored in string variables may assume the full 8-bit ASCII character code range from 0 to 255. String variables in APCBASIC may be defined to hold any length string that the memory in your machine will permit. However since strings are variable length objects, APCBASIC sets aside an area for each string variable that will hold any string up to a maximum length. Unless you inform APCBASIC how large a particular variable should be, APCBASIC will assign a default area that can hold only up to 10 characters. You may assign your own maximum string size using a DIMension statement like this:

```
DIM A$(50),B$(9999),C$(1)
```

where A$ may store 0 to 50 bytes, B$ may store 0 to 9999 bytes and C$ can store only 1 or 0 bytes. The same DIM statement can define one or more strings by listing their definitions one after another, separated with commas as shown above. Both string and numeric (array) variables may appear in the same DIM statement. Newly DIMensioned strings are filled to their maximum length with spaces (ASCII 32). This default may be altered at any time to any ASCII code from 0 to 255 using PARAM(7) (Section 5.8).

DIMensioning a string that already exists is legal everywhere except in FNs and FOR..NEXT loops. Such an operation is useful for releasing unneeded memory back to the system for further use, and to permit program control over the size of string and array variables. Since DIMensioning always re-initializes strings (with the default ASCII code), all previous contents of the variable are lost (as in numeric arrays).

— 4.3 STRING ARRAYS —

Another type of string variable is the array, in which a group of string values can be stored under one name. String arrays are organized as an ordered set of storage locations, called array elements, that are identified by a position number within the ordering. For example A$(0), A$(1) and A$(2) represent the first three string elements of array A$(). Parentheses are used to indicate that A$() is an array and serve to contain the position of the desired array element. The positions are sequentially numbered from zero up to the size of the array.

The 1-dimensional array A$() above could act as storage for a list of lines of text, collectively representing a page of text. Thus you can directly access each line on the page by its line (position) number. Suppose that we combine many such pages together into one string array for access by page (position) number. This is called a 2-dimensional array. By identifying the line and the page we can directly access any line in the 'volume'. For example A$(I,J) refers to line J on page I, where I,J are simple variables specifying the array element positions. By adding further levels to this idea, you can define and access string arrays with 3 or more dimensions. An N-dimensional array requires N position numbers, called subscripts, to uniquely specify an element position in the array.

DIMensioning String Arrays

In order for an array to exist it must be defined in your program prior to its use. The definition of a string array must include its name, a maximum position for each dimension subscript, and the string capacity of each of the array elements. Specify string array DIMensions just like numeric arrays except that you must include the maximum length of each array element as the last value of the DIMension list. Take the following 2-dimensional string array definition example:

```
DIM B$(7,20,16)
```

This defines an 8 by 21 (zero-based array subscripts), two-dimensional string array whose array element strings may range from 0 to 16 characters each. You must always refer to B$ with a subscript list to indicate a specific array element, as in R$=B$(I,J); R$=B$ would generate a SYNTAX ERROR. When accessing string array elements, specify only the array DIMension positions and leave off the length parameter, given only when DIMensioning.

An OUT OF BOUNDS ERROR occurs if you attempt to access a dimension position outside its defined range. A single DIM statement can define one or more arrays by simply listing their definitions one after another separated by commas. All array elements are initialized the same way as simple strings. You cannot assign the same string variable name to both a string array and a simple string variable. All string arrays must be defined explicitly, otherwise APCBASIC thinks they are simple string variables instead of arrays.

You can re-DIMension the array at any time (except within FNs and FOR loops) by re-defining it in another dimension statement. All stored strings are erased after such an operation and re-initialized. In this manner arrays can grow or shrink depending on your program requirements. When arrays are made smaller the unused memory space is available to the system for use elsewhere. Never re-DIMension anything inside FOR..NEXT loops (Section 5.4) or inside User-DEFined Functions.

— 4.4 STRING EXPRESSIONS —

String are manipulated and processed by combining them in structures known as string expressions, not unlike numeric expressions. Such expressions permit you to specify a string as a combination of other strings and are formed from string symbols and string operations. Although the notation of string expressions looks similar to numeric expressions, their operation is totally different. For example the string expression "ABCDE"+"12345" evaluates to the new string result of "ABCDE12345". As you can see, the plus sign (+) has a different meaning depending on whether it is being applied to numbers or to strings.

String symbols used in string expressions include string constants, string variables, string functions (including user-defined FNs), and string sub-expressions. A sub-expression is actually a portion of a larger expression that has been surrounded by parentheses, grouped as a computational unit.

String operations, called string operators, are of two types: unary and binary. Unary operators act on a single string to form the result string. For example the NOT operator preceding a string (eg. NOT Z$) will produce a result string of the same length but with each byte logically complemented. Binary operators however act on two strings situated on either side of the operator to combine them in some fashion producing a result string, as in the plus sign operator demonstrated above.

APCBASIC evaluates string expressions from left to right accumulating the results from each operation as it goes. The various string operators are not however applied with equal priority. Take for example the following string expression involving concatenation (plus again) and string repetition (*) factors:

```
"ABC" * 2 + "xyz" * 3
```

This expression produces a result consisting of "ABC" repeated twice and followed by "xyz" repeated three times (ie. "ABCABCxyzxyzxyz"). Since the string factors (*) are evaluated before the concatenation, we say that such factors take precedence over concatenation (just like their numeric counterparts). Similarly, all string operators have been assigned to a priority scale that controls the order of operations when several precedence levels are present in the same expression (much like the numeric operator precedence ordering).

When required, you can override these default priorities by surrounding any operation by parentheses to force its evaluation in the order of your choice. The example below illustrates a situation where concatenation (+) is performed prior to a string repetition factor:

```
("ABC" + "xyz") * 5
```

The concatenation in parentheses is evaluated first, followed by repeating its result five times. The table below lists the various string operators in order of decreasing precedence followed by a discussion of each.

STRING OPERATOR PRECEDENCE TABLE

(0) Evaluate string constants, string variables, string functions and sub-expressions.
(1) String Indexing (Section 4.5)
(2) NOT
(3) String Repetition Factors (*)
(4) String Concatenation (+)
(5) Comparision Operators (=, <, >, <=, >=, <>)
(6) AND
(7) OR
(8) XOR
(9) EQV

This ordering is similar to that of numeric expressions except that strings have fewer operators. Any ordering of operations may be accomplished using appropriately placed parentheses. Be most careful in using complex string expressions in string comparison operations. The comparison operators are not really string operators since they produce a numeric result (0 or 1). They are included in the table above only to show their precedence within mixed mode expressions. It is the programmers' responsibility to ensure that mixed string and numeric expressions are sufficiently parenthesized to resolve any inherent ambiguities.

STRING CONCATENATION

The simplest of the string operations is concatenation (+), which merely appends two string operands together, end to end, in the order given. For example "ABCDE"+"12345" = "ABCDE12345".

STRING REPETITION FACTORS

Any term of a string expression may be repeated by following the term by a multiply operator (*) and a numeric expression [eg. "ABC" * (X+Y)]. First the factor is evaluated (X+Y), then the string is repeated by that many times. The repetition factor expression needs parentheses surrounding it only if it contains more than one numeric term, as in the example above. When only simple factors are used, no parentheses are required, as in the string expression:

```
A$ * X + B$ * 37 + C$ * 23.
```

Any complex string expression may be multiplied by enclosing it in parentheses followed by the desired multiplier [eg. (A$+STR$(N)+"XYZ") * (R+2)]. Compound nesting is permitted to almost any depth. Typical applications of string multiplication include dynamic formatting of strings in print statements, high-speed graphics, and initialization of large strings. String expressions are always formed in APCBASIC's control stack, which can rapidly overflow when compound repetition factors build up enormous strings that exceed the available memory space.

LOGICAL OPERATORS IN STRING EXPRESSIONS

Logical string operators (NOT, AND, OR, XOR, EQV) perform processes similar to their function in numeric expressions, except that both operands and result are bit-strings. NOT performs a logical reversal on each bit of the operand string following it (1s become 0s, 0s become 1s). Its result string is the same length as its operand.

All other logical string operators are binary (dual-operand) and result in a string which is a logical combination of corresponding bits in the operands. If the operand strings differ in length, the shorter of the two will determine the extent of the operation and hence the length of the result. See the truth table of numeric logical operations in Section 3.4 for the definition of these logical operators, remembering that the operands and result are bit-vectors.

This type of processing is exactly suited to processing set data structures. AND and OR implement the Intersection and Union operations respectively, while NOT produces the complement of a set. Set operations have many applications. For example in data base applications, sets of various item selections may be logically combined into a single set of items representing a complex selection.

Other useful applications for bit-vector operations include the following conversion from lower case to upper case. It turns out that if you set bit5 of a character to the logical combination of (NOT bit6 AND bit5) then the resulting character will be upper case (see an ASCII code chart to verify this as an exercise). This operation can be performed on an entire string using the following string assignment statement:

```
U$ = NOT ROTAT$(L$ AND CHR$(64)*LEN(L$),1) AND L$
```

where L$ is the original string, U$ is the upper case result string, and LEN(), CHR$() and ROTAT$() are string functions described in Section 6.3. A similar statement may be implemented to convert from upper case to lower case. If often required within your program, this is best programmed as a User-Defined String Function (one-line function).

COMPARISON OPERATORS

Comparison operators are different from all the other string operators in that they give a numeric result instead of a string result. This result is always a floating point zero or one, just as if the two operands were numeric values being compared. Such an operation must therefore be in the context of a numeric expression. Each comparison operator compares its operands and returns TRUE or FALSE (1 or 0) to indicate the outcome of the comparison. Both operands must be of the same data type (attempting to compare a number with a string results in a SYNTAX or TYPE error).

OPERATOR	DESCRIPTION
=	Equality
<	Less-Than
>	Greater-Than
<=	Less-Than or Equal
>=	Greater-Than or Equal
<>	Not Equal

When strings are compared, the ASCII codes of corresponding characters are compared from first to last until an a difference is detected or the end of either string is encountered. Strings are equal only if all characters are identical and both strings are of equal length. If one string 'runs out' before a difference is encountered the longer string is defined as 'greater than' the shorter string. It is important to remember that string comparisons give logical (0 or 1) results which may be used anywhere that numbers are permitted.

Exercise great care when complex string expressions are supplied as comparison operands. String operators look similar to numeric operators but their actions are totally different. APCBASIC requires the programmer to resolve any such ambiguities by placing parentheses () around ambiguous sub-expressions. A simple example is the expression: X+A$>B$. APCBASIC trys to evaluate this as (X+A$)>B$, whereas its only meaningful usage is X+(A$>B$). In this example APCBASIC will generate a TYPE ERROR for the first expression when encountered. However is it quite possible to construct more complex expressions that execute witnout detectable errors but produce nonsensical results.

— 4.5 STRING INDEXING —

Although a string is composed of separate characters, we have up to now been treating strings as indivisible units of information. It is often desirable to access portions of strings rather than their entirety. Most programming languages implement such access through special functions like LEFT$(), MID$(), RIGHT$() and SUBSTR$(). APCBASIC uses a different method that is easier to learn, executes faster and performs the job in a more general fashion.

By convention, the beginning and end of a string (oriented horizontally) shall refer the its left and right ends respectively. String indexing is based on the idea that each character in a string has a position relative to the beginning of the string. Let the first character be in position 1, the 2nd character in position 2 and so on to the end of the string. Any portion of a string could then be designated by a position range within the defined positions of the string. For example if A$ is our string and we wish to access positions 10 through 27, we would express this as follows:

```
A$(10,27)
```

As long as the length of A$ is 27 or more, this indexing expression accesses the 18 characters in A$ starting at the one in position 10. If A$ contains less than 27 characters, APCBASIC will access all characters from position 10 to whatever the length of the string. A null string ("") results if A$ is less than 10 characters long. Any string constant, string variable, string function or sub-expression may be the subject of an indexing expression.

Variations on this theme provide several other modes to specify substrings in diffent ways having advantages over one another. Each of the the string indexing modes are discussed in the table below. The examples shown in the table use the variable A$ to represent a general string expression to which the indexing expression is applied.

String Indexing Modes

MODE	EXAMPLE	D I S C U S S I O N
Interval	A$(I,J)	A$(I,J) refers to the substring starting at position I and ending with the byte at position J (inclusive).
Open Ended	A$(I)	Refers to the substring consisting of all bytes from position I to the last byte of the string.
Position & Length	A$(I:L)	Refers to a string of length L starting with the byte at position I. This is equivalent to A$(I,I+L-1) using the Range method. A null string results if L=0.
Right Length	A$(:L)	Refers to a string of length L taken from the end of A$. Equivalent to A$(LEN(A$)-L+1:L).
Single Byte	A$(I:)	Refers to the single character substring in position I of string A$. Equivalent to A$(I,I) or A$(I:1).
Last Byte	A$(:)	Refers the single character substring at the end of A$. This follows from the preceding two indexing modes as a special case. Equivalent to A$(LEN(A$)).

Given a string A$(I,J), APCBASIC returns a null string ("") whenever J is less than I (J=0 is permitted) or I is greater than the length of A$. Also if the substring specified exceeds the length of the stored string, only that portion which is defined will be accessed. For example if A$="This is a String", then A$(9,1000) = A$(9:100) = A$(9) = "a String". Examine these carefully to see how rules (1)-(3) apply. An OUT OF BOUNDS ERROR occurs on a starting position less than 1.

Assigning Strings to Indexed String Variables

An indexed string variable may be target of an assignment statement or any other operation that moves data into a string variable. However such an assignment can only affect defined character positions within the indexed region specified and cannot alter the overall length of the string. Strings moved into these positions are truncated (from the end) when to long to fit. Shorter strings are placed left-justified into the indexed area, replacing only those characters in positions required by the incoming string.

String arrays may be indexed by following the array subscript expression with a string indexing expression (2 sets of parentheses). In such a case, you are gaining access to a substring in an array element of a string array. String array elements are always functionally identical to simple string variables in any context.

Extended String Indexing

Index expressions may be appended to any string representation, including another indexed string. This flexibility permits several layers of indexing to be applied to the same string, which can facilitate implementation of various hierarchial data structures stored in large string variables. For example:

```
A$(I,J)(R:L)(T)
```

Each indexing expression is evaluated from left to right and is applied as a simple indexing expression to the result substring of the prior indexing expression. Internally, APCBASIC arithmetically evaluates a series of indexing expressions as a unit and only then does it apply it to the string being indexed. This replaces many potentially time-consumming string move operations with a simple binary arithmetic computation that executes many times faster. At the cost of some floating point arithmetic, this same example could have been done with a single indexing expression as follows:

```
A$(I+R+T-2,MIN(J,I+R+L-2))
```

Not only does this approach execute more slowly, but it is not at all obvious what is really going on. Extended string indexing simplifies certain kinds of operations but in the vast majority of applications simple indexing should be all that is necessary.

— 4.6 STRING FUNCTIONS —

Strings can be represented in expressions not only as constants and variables but also as results of special procedures called functions. For example REV$(A$) is a built-in function that returns A$ in reverse order. Functions are always of the same form: <function name> (<argument list>). In the REV$() example, the argument list consists of one parameter: A$. Enclosed in parentheses, the argument list contains data expressions (separated by commas) which the function uses to produce a single string or numeric result.

Arguments may be string or numeric expressions whose number and type depends on the particular function being used. APCBASIC possesses a library of about fifty built-in functions and provides facilities for user-defined single and multiple line functions.

Section 6.0 provides a complete description of all the built-in functions in APCBASIC. These should be studied for utility in your particular applications since there may be several that already do what you have in mind and they run many times (even hundreds of times) faster than similar procedures written out in APCBASIC. They are divided into arithmetic functions, mathematical functions, string functions (including bit operations), file and device I/O functions, and system interface functions.

The division between string and numeric functions become a little unclear when arguments and result are a mixture of string and numeric data types. For example LEN(A$) returns the numeric length of A$, is it a string function? Although such functions are associated with strings, a function which returns a string will be referred to as a string function in this manual unless otherwise noted.

There is no significant difference between User-DEFined string functions and User-DEFined numeric functions. String function names are formed by concatenating "FN" with a valid string variable name (similarly with numeric functions). A string result is returned from a string function, a number is returned from a numeric function. Both may be defined with string or numeric parameters and formats include both single and multiple line. See Section 3.5 for all the details on defining and using User-DEFined functions.

**** 5.0 APCBASIC PROGRAM STATEMENTS ****

This section provides brief descriptions of all executable statements available in APCBASIC. See Section 2.0 for the description of the notation used to specify command and statement formats also employed in this section. The statements are grouped into the following categories:

5.1 Data Definition	Setting sizes, providing memory space, establishing initial values for working variables and defining data constants for program operation.
5.2 Input/Output	Interactively exchanging data with the user, formatting results and editing during data entry.
5.3 Data Transformation	Moving data between variables, packing and unpacking bit-strings and performing computations.
5.4 Program Control	Altering the sequence of program statement execution, looping structures and error control processing.
5.5 Subroutines	Defining and Executing subprograms and providing temporary local storage for their computational needs.
5.6 File Processing	Accessing files and their data.
5.7 Segmentation & Overlays	Executing programs larger than the the memory space available.
5.8 System Interface	Altering APCBASIC system parameters, direct access to memory and I/O ports.
5.9 Documentation	Including descriptive remarks within programs.

-- ALPHABETICAL STATEMENT SYNTAX SUMMARY --

```
<numeric variable> = <numeric exprn>
<string variable> = <string exprn>
BIT(<string variable>,<bit address>[:<bit width>]) = <exprn>
CALL <address exprn>,<data register exprn$>[,<result register vbl$>]
CLOSE [#<file number>]
CREATE <new file name>,<file size>[,<file type>]
DATA <data list>
DEF <function name>[(<argument list>)][=<expression>]
DELETE <line# range>
DESTROY <existing file name>
DIM <list of string & array definitions>
DIR [#<device number>,] [<drive number>]
DOS
EDIT <string exprn>
END
ERRSET #<numeric error type exprn>
ERRSET [<line# trap> [,<error line variable> [,<error type variable>] ] ]
EXAM <starting address>,<variable list>
EXIT [<line#>]
FILL <starting address>,<data list>
FNEND
FOR <index variable> = <range1>,<range2>,<range3>,...,<rangeN>
FREE
GOSUB <line#>
GOTO <line#>
IF <logical exprn> THEN <statement1> [ELSE <statement2>]
INPUT[1] [#<device number>,]<input list>
LINK <program name string exprn> [,<common variables>]
LOCAL <list of string & scalar variables>
MERGE <program name stringexprn>
NEXT [<index variable>]
NOMARK <logical exprn>
ON <exprn> GOSUB <line# list>
ON <exprn> GOTO <line# list>
ON <exprn> RESTORE <line# list>
OPEN #<file number> [% <file type>], <file name> [, <size variable>]
OUT <port number>,<8-bit data value>
PARAM(<exprn>) = <exprn>
PRINT [#<device number>,]<data list>
READ #<file number>,<data variable list>
READ <list of data variables>
REM <descriptive text>
RENAME <old file name>,<new file name>
RESTORE <list of variables>
RESTORE [<line#>]
RETURN [<exprn>]
STOP
SWAP <list of variable pairs>
SWAPDEF <list of variable pairs>
WHILE <logical expression>
WRITE #<file number>,<data exprn list> [,NOMARK]
```

DIM <list of string & array definitions>

Sets aside memory space for simple strings, string arrays and numeric arrays. DIM is an executable statement, so space for its defined variables will not be allocated until the DIM statement is actually executed in the running program. You must therefore make sure that your DIM statements are executed before the variables they define are used in any way by your program. If a variable already exists, DIMensioning it will alter its size and re-initialize it as if it was being created for the first time. See Sections 3.2 and 4.2-4.3 for a complete discussion of DIM, arrays and strings.

RESTORE <list of variables>

Restores (or reinitializes) the variables listed to their original contents at creation time (by default or DIMension statement). Specify arrays and strings the same way as LINKs passing variables. Simple numeric variables may be listed by name. Strings are always filled to their DIMensioned size with blanks unless the default string initialization ASCII code has been modified using PARAM(7) (see Section 5.8). Numeric variables including arrays are filled with zeros. For example: RESTORE X,Y,A(),B$,R$(), where the empty parentheses indicate array variables. This use of RESTORE is unrelated to its use with DATA-READ statements.

DATA <data list>

Specifies a list of numeric and/or string (in quotes) constants, separated by commas. DATA statements do nothing when encountered during program execution, as their purpose is solely to provide programs with built-in data values which may be assigned to variables by the READ statement (described next). DATA statements are unsuitable for large amounts of data or when the data will be revised during program execution. In such cases you should store the data on disk files for the purposes you have in mind.

DATA statements in a program are best visualized as a list of statements separate from the rest of the program, but in the order given in the program. Data supplied in DATA statements is accessed sequentially from the beginning DATA statement or from a starting line number given by a RESTORE statement described below.

READ <list of data variables>

Sequentially READs string or numeric constants from the current DATA statement into the list of data variables. Data variables may include simple and array numeric variables and both unindexed and indexed strings. If the current DATA statement runs out of data before filling all data variables, then the next DATA statement in the program is automatically found and READing continues to the end of the variable list. Both DATA and READ lists are scanned in the order given, and all variable types must match the data items encountered. An error results from a type mismatch or from an attempt to READ past the last DATA statement in the program.

APCBASIC maintains an internal READ pointer to keep track of the current DATA position. This pointer is set to the first DATA statement when program execution starts. Thereafter, the pointer is localized within GOSUBs & user-defined functions (DEF FN...) so that DATA lists may be used in subroutines without affecting the access to data lists outside the subroutines. This is more powerful than the 'global' READ pointer supported in standard BASIC, but may not be compatible with some programs that use DATA statements in standard ways. For example, GOSUBs cannot be used to set up the READ pointer for subsequent READ operations after its RETURN. After such a RETURN, the pointer will always appear unchanged from where it was before executing the GOSUB.

RESTORE [<line#>]

Sets the DATA READ pointer to the first DATA statement in the program, or to the first DATA statement after the requested line number. This is used to reset the READ pointer or to provide random access to DATA statements. This use of RESTORE is unrelated to restoring variables described earlier in this section.

ON <exprn> RESTORE <line# list>

Evaluates the numeric expression and truncates the result to an integer that specifies a line number in the <line# list>. This integer must be from 1 to the length of the <line# list>. Then the DATA READ pointer is set to the first DATA statement on or after that line number. This is useful for selecting data through a multi-way computational decision.

— 5.2 INPUT/OUTPUT STATEMENTS —

These program statements allow the transfer and formatting of data between your APCBASIC program and 16 virtual devices (numbered from 0 to 15). All devices are considered general purpose byte-oriented I/O data channels whose details are the concern of the operating system. By convention devices 0 and 1 represent the console and the printer (or list device) respectively. Device numbers 0-7 refer to actual devices, while numbers 8-15 access open files when required (see below). CP/M versions have the option of patching jump vectors to map devices 0-7 to arbitrary machine I/O drivers present somewhere in memory (using the CONFIG program described in Section 6.3). See Section 7.4 for further details on the CP/M device I/O channels.

Devices 8 through 15 may be employed to connect I/O operations with files OPENed under any corresponding file numbers. This usage allows files to appear as though they were ordinary devices, allowing redirection of PRINTed material to files, INPUT of strings from text file, or LISTing programs to ASCII text files. Remember that a file must be OPEN under a 'device' number in the range 8-15 before such I/O operations are attempted. See further details under the specific statement or command you wish to apply (any process that directs I/O using #device numbers). See Section 5.6 for further details on file processing (especially OPEN, CLOSE, and NOMARK).

In all cases, device I/O statements may optionally specify any of the 16 device numbers to access that device. When omitted, APCBASIC accesses the current default device number. Initially zero (#0 accesses the console), this default can be changed at any time by using the PARAM(3)=<exprn> statement described in Section 5.8.

PRINT [#<device number>,]<data list>

Causes the data list specified to be output as characters on the device specified (or the default device). The <data list> is a series of numeric expressions, string expressions, format specifications and control specifications separated by commas. The numeric and string expressions are evaluated and PRINTed in the order in which they appear, while the format specifications describe how subsequent numeric data will be presented on the output device. Control specifications perform useful operations during the printing process such as tabs and extra blank lines. Because of the frequent use of PRINT statements in most programs, the 'PRINT' keyword can be replaced by an exclamation mark (!) for brevity. This notation performs the identical function that PRINT does.

APCBASIC scans the data list, printing numbers and strings as they come and attending to format and control specifications as encountered. Numbers are printed in the currently defined format, which may be redefined any point in the data list. At the beginning of the print statement, the currently defined format is the default format, which may also be re-defined at any point.

Normally a carriage return is generated on completion of a PRINT statement, but this can be suppressed by terminating any PRINT with a comma. This permits several PRINT statements to contribute to the formation of a single line.

Numeric Format Specifications

Format specifications always begin with a percent sign (%) to indicate that a format follows. By itself, this specifies free-form formatting (described below). Other formats are denoted explicitly (after the percent %) in the form: wXr, where w=field width (total number of characters), X=mode character (I, F, or E), and r=number of trailing decimals (to the right). Such numeric formats position the value right-justified within the specified field width, ideal for single or multiple columns of numbers. Omitting the width (w) formats the number with one leading space, r trailing decimals (E,F only) and no trailing spaces.

Under free-form format, which is also the initial default format, APCBASIC outputs numeric values as follows: a single space, a minus sign (-) if negative, followed by the value itself including all the digits of precision available (no trailing zeros after the decimal). When the value cannot be represented without shifting the decimal with leading or trailing zeros, the value is formatted with E notation (all digits of precision supplied). Note the precision of your APCBASIC determines where the switch-over occurs between E-notation and standard formats. Each of the numeric format specifications is described below:

wI — Right-justifies an integer in a field w columns wide. No trailing decimals (r) can be specified. Non-integral values are rounded prior to printing. Free-form integers are generated by omitting the width (w).

wFr — Right-justifies a number with r places to the right of the decimal within a field w columns wide. Free-form floating point layout is formed by omitting the width (w).

wEr — Same as wFr except that E-notation (scientific) is used. This notation prints as a base value (X.XXXXX...) followed by a power-of-ten scaling factor (E+XX or E-XX) called the exponent. The exponent always appears as the last 4 characters of the specified field. See Section 3.1 for a discussion of E-notation.

Attempting to PRINT data using an insufficient width results in the entire field being filled with asterisks (*), instead of a number (a programming error in your program that must be rectified). Both w and r, when supplied, must be given as unsigned integers. Only one numeric format (I, F, or E) can be included per specification. When r=0 (zero decimals), the number is rounded to an integer and the decimal point removed.

Format Modifier Specifications

Each of the numeric specifications may be preceded by certain format modifiers to produce additional features: dollar signs, comma grouping (eg. 1,435,801), zero suppression, etc. Such modifiers consist of a single character which when inserted in front of a numeric specification invoke the desired effect. When not followed by one of the format specification described earlier, the modifers act on free-form formatting. All modifiers are described below:

$ Places a dollar sign ($) to the left of each number printed. When a leading numeric sign (+ or -) appears with the dollar sign ($), the sign comes first, then is followed by the dollar sign. Be sure to provide sufficient room in your field widths (w) allowing for the dollar sign. See the special note concerning ambiguity in the discussion of dynamic format specifications later on.

C Inserts commas every three places (left of the decimal) after 1000. Remember that these commas take up space in your specified widths. This modifier (C) has no effect on E-formatted numbers.

Z Suppresses trailing zeros to the right of the decimal. Trailing zeros are changed to spaces (blanks). If the format does not include a width specification (w), then these spaces will not appear in the field. The 'Z' modifier has no effect on I-formatted numbers.

+ Indicates positive numbers with a plus sign (+), the same way as negative numbers are shown with a minus sign (-). All numbers will be printed with a numeric sign, regardless of their value.

T Positions the plus (+) or minus (-) sign to the right of the value (trailing sign), instead to the left (leading sign). The sign will appear as the last character of the specified field, which might not immediately follow the last numeric digit (see the 'Z' modifier above). Be sure to account for the additional place after the decimal in your printing layout (the width is unaffected). 'T' has no effect on E-formatted numbers.

D Specifies all modifiers that are currently defined in the default format. All immediately preceding format modifiers are lost and thus this modifier should be first when more than one is supplied.

Causes this format specification to become the default format as well as the current format. This is not really a format modifier since it has no effect on the current format. Used alone in the specification, # sets the default format to free-form numeric output.

Several modifiers may be listed between the % and the numeric specification for their combined effect. If no numeric specification follows the modifier(s) then a free-formatted number will be modified accordingly.

Dynamic Format Specifications

Dynamic formatting permits your program to determine the format specification at run-time rather than being fixed throughout execution. Such formats are specified by following the format lead-in character (%) with a string expression instead of the usual format specification. This expression must evaluate to a legal format specification, which is then applied. For example to print X with comma-insertion, zero-suppression, dollar sign, right-justified in a field W characters wide with D decimals, use the statement:

```
PRINT %"$ZC"+STR$(W)+"F"+STR$(D),X
```

Because of this flexibility, you must watch out for non-dynamic specifications that 'look' dynamic. For example does the statement 'PRINT %C$,X,Y,Z' specify dollars with comma-insertion or does it specify a dynamic format in the string variable C$? This is easily rectified by placing quotes around such specifications (%"C$") or by reordering the format modifiers to eliminate the ambiguity (%$C). APCBASIC always assumes a dynamic format if what follows the percent lead-in (%) 'looks like' a string expression. Such ambiguities can only occur when the dollar sign is used as the 2nd character of the format.

Control Specifications

Special control specifications may also appear in the <data list>. These are not format specifications and are not preceded by a percent (%). A plus sign (+) resets the line count for the device to zero before proceeding and is necessary only in applications where this count is being used with the LINES() function. The plus sign (+) does not generate any printed characters and has nothing to do with the similar format modifier (+).

Multiple blank lines may be generated from a single PRINT statement by a field of slashes, similar to FORTRAN format statements. For example: PRINT #D,///, will generate 3 carriage returns on device D. Slashes may be interspersed throughout the data list.

APCBASIC generates a carriage return at the end of the PRINT statement unless you suppress it by ending the statement with a comma. For example 'PRINT X' will display X followed by a carriage return, while 'PRINT X,' displays X and suppresses the carriage return so that later PRINT statements can continue PRINTing on the same line.

Finally the TAB(P) function may appear to advance to column position P prior to PRINTing the next item, where P is a numeric expression that evaluates to a value from 0 to 255. This is accomplished by PRINTing spaces until the desired position is reached. TAB(P) is ignored if P is less than or equal to the current position.

Printing on Files

When PRINTing to 'devices' 8-15, you are really transferring data to a file OPENed under the same file number. Exactly the same data is transferred to the file as would be displayed on the console if device #0 were employed. However it is important that the last byte PRINTed before the file is CLOSEd be an appropriate end-of-file mark so that the file can be processed correctly by other programs. The APCBASIC endmark (a 8-bit value of 1, ASCII CTRL-A) is placed automatically after each PRINT for this purpose.

This is appropriate for later INPUT processing by APCBASIC programs, but typical text-oriented programs external to APCBASIC usually expect other endmarks. For example ASCII codes 0 and 255 are common and CP/M environments require an ASCII code 26 (CTRL-Z) to terminate text files. You must handle this situation by PRINTing the appropriate endmark as the very last byte written to the file as dictated by these external requirements. Note that whatever code is used cannot be part of the text PRINTed without causing a false end-of-file condition when later processed.

INPUT[1] [#<device number>,]<input list>

Inputs data from device 0-15 and it stores into a list of string or numeric variables. All devices use the same old-line buffer through the APCBASIC line editor. Each input must be entered with a carriage return as the last character and must be less than 160 characters long (2 screen lines). If this input format is undesired, use the INP() or INCHR$() functions described in sections 6.5 and 6.1 respectively. APCBASIC echos a carriage return after each input entry, but this may be suppressed with a 1 (for one-line input) placed immediately after the INPUT keyword.

The input list contains variables which receive the input data and whose type (string or numeric) determines how the input characters are to be interpreted. By entering a numeric constant (followed by a carriage return) you store a value into a numeric variable, or a string that 'looks like' a number into a string variable. By entering a line of numeric constants separated by commas (and optional spaces) you can input several numeric variables at once, and any prompts encountered for these inputs (after the initial prompt) will be ignored. String variables accept the entire line of input, even if it contains spaces, commas, numbers or words and phrases. Input strings larger than the DIMensioned size of the input variable will be truncated to fit the string.

The input list may also contain string messages called prompts, which are displayed on the device to signal or request specific input data. Normally, APCBASIC prompts you with a question mark (?) for each variable input. However you can insert your own prompt (string expression) in front of any input variable. Remember that it must not begin with a variable name and it must be followed by a comma and the input variable(s). Thus a complete input list would be a series of: prompt1, vbl1, prompt2, vbl2, ..., promptN, vblN. To suppress all prompt messages, including the automatic question mark, specify a null string ("") as your prompt. This permits prompts from PRINT statements elsewhere in your program.

When INPUTing from 'devices' 8-15, APCBASIC takes data from a file instead of the usual device and enters a special mode of operation. All prompts are ignored and only string variables may be specified to receive input (a TYPE ERROR occurs on encountering numeric variables). Carriage returns and line-feeds (ASCII codes 13 and 10) are treated differently when they appear in pairs. A carriage return line-feed sequence collapses into a single carriage return code (the line-feed is ignored). A line-feed carriage return sequence collapses into a single line-feed (the carriage return is ignored).

Strings from devices 8-15 are INPUT as a sequence of characters ending with a carriage return or an end-of-file mark (8-bit value of 1), whichever comes first. This ending character serves only to define the end-of-line and is not otherwise a part of the INPUT data. Thus a sequence of carriage returns is INPUT as a sequence of null strings (""). A TYPE ERROR occurs if your program attempts to INPUT a string when the first character is the end-of-file mark (1 code). You can test for this condition by examining the TYP(N) function for a value of zero before each INPUT line (see Section 6.4), or by trapping the error with the ERRSET statement (see Section 5.4).

EDIT <string exprn>

The EDIT command may be executed within a program to allow editing a string not previously entered. Used in conjunction with and prior to an INPUT statement, EDIT places the string expression evaluated into the old-line buffer so that editing CTRL characters may be used on it. Very powerful tool for editing any data already within the grasp of your program. For example the statement sequence:

```
EDIT A$+B$+C$; INPUT "Edit string -- ",R$
```

This permit editing of the string produced by the concatenation of A$, B$ and C$ without prior entry of that string from the keyboard.

— 5.3 DATA TRANSFORMATION STATEMENTS —

These statements permit transfer of data between variables and support combinations of many data items into single data results through arithmetic, mathematical, logical, or other computational means. You should understand how to use variables and constants (Sections 3.1-3.3), strings and string variables (Sections 4.1-4.3), string and numeric expressions (Sections 4.4-4.5 and 3.4), and functions (Sections 3.5 and 4.6), to make full use of the data transformation statements.

<numeric variable> = <numeric exprn>

Assigns the numeric value evaluated from the expression on the right of the equals sign (=) to the numeric variable on the left. The variable may be a simple variable or a unique array element. Note that this (context dependent) use of the equals sign has nothing whatsoever to do with its use in equality comparision expressions.

<string variable> = <string exprn>

Assigns the string generated from the string expression on the right of the equals sign (=) to the string (or sub-string) variable on the left. If the string variable is not indexed then its entire previous contents is replaced by the string on the right and that may change its string length. If the string variable is indexed then only its indexed sub-string portion is affected. Strings longer than this sub-string are truncated to fit and shorter strings are placed left-justified within the sub-string field. This indexed mode of assignment cannot alter the length of the string.

Care should be taken to avoid assigning expressions involving very long or many strings in one expression, since working memory proportional to the length assigned is required. Use the FREE() function (Section 6.5) to determine how much memory is available when such operations might cause an MEMORY FULL ERROR in order to perform the procedure a different way (possibly by breaking it up into smaller pieces that can be processed independently).

SWAP <list of variable pairs>

Exchanges the contents between each pair of variables listed. Both variables of each pair must be of the same type (string or numeric). Each variable pair is exchanged independently of one another. The list of pairs is processed from left to right and may contain a mix of string and numeric variables. For example:

```
SWAP X,Y,A$,B$,R4(J),Z(I,K)
```

A SYNTAX ERROR results from an odd length list. A TYPE ERROR results from attempting to SWAP strings with numbers. SWAP is limited to single string or numeric data. To swap arrays, you can use the SWAPDEF statement below.

When exchanging the contents of two numeric or string variables, the SWAP statement may be used to gain a 3-5 times speed improvement over the usual assignment statement implementation. SWAP is directly useful for sorting routines, where total sorting time can be cut substantially. When strings of differing lengths are SWAPped, APCBASIC uses the shorter of the two lengths as the length of the swap; characters after the SWAPped area of the longer string remain unaffected. Hence if one of the strings is null ("") then no action is taken.

SWAPDEF <list of variable pairs>

Exchanges variable definitions between variable names given in each pair. The variable pair list follows the same rules as the ordinary SWAP statement except that array variables must not be subscripted and string variables must not be indexed. Instead, simple string or numeric variables are always specified by name only and arrays (both string and numeric) are indicated by their name followed by an empty set of parentheses, for example:

```
SWAPDEF X,Y,A$,B$,X(),Y(),D$(),E$()
```

Suppose that X() and Y() two numeric arrays that are dimensioned 5x5 and 8x8x10 respectively, and each contain data. After the statement SWAPDEF X(),Y() has been executed, X() is now the 8x8x10 array and Y() is the 5x5 array. SWAPDEF merely re-labels the variables of each pair as each other.

Because the data in the variables is not actually moved (as done by an ordinary SWAP statement), the execution time required is extremely small and independent of data size. SWAPDEF may be employed to pass arrays and strings to GOSUBs (or FNs) by easily substituting them for the variables used within the GOSUB procedure, then SWAPDEFing them back again after the RETURN. The variables are indicated under the same conventions for specifying a LINK common variable list.

BIT(<string variable>,<bit address>[:<bit width>]) = <exprn>

Evaluates the numeric expression (on the right), converts it to a non-negative integer, and assigns the result to any bit sequence (1-16 bits wide) within a string variable. The string variable reference may be indexed or unindexed; the <bit address> is a numeric expression that specifies the relative bit position within the string at the start of the bit sequence. The (optional) <bit width> expresses the number of bits to be stored, which defaults to 1 when omitted. BIT() may appear on either side of the equals sign depending on whether you are storing a value (left) or accessing a value (right).

The BIT function is capable of accessing groups of 1 to 16 bits as a numeric unit. This provides very efficient utilization of memory when large tables of small positive integers are required. The <bit address> specifies the 1st bit of the group, which must range from 0 to (string length * 8 - width) and cannot exceed 65535. Values are stored modulo 2^width, causing reduction of values too big for the given length. Widths outside the 1-16 range generate an OUT OF BOUNDS error. For example BIT(A$,17:8)=259 stores a value of 3 into the 8 bits starting at bit 17 of A$.

The <bit address> always refers the the high-order bit of the integer (bit sequence) being accessed. The table below illustrates the relationships between BIT addresses, bit numbers (within bytes), and memory byte addresses. For simplicity we assume the 1st byte of the string to be located at memory address 0.

```
Byte Address:  0        1        2        3        4        5        6
Bit Numbers:   76543210 76543210 76543210 76543210 76543210 76543210 76543210
BIT Address:   0  4     8  12    16 20    24 28    32 36    40 44    48 52
```

For example bit4 of the 3rd byte of the string has the BIT address of 19. Note that the BIT addresses in the table go by fours only to simplify illustrating the idea.

-- 5.4 PROGRAM CONTROL STATEMENTS --

Normally, program execution proceeds sequentially through the statements in order by line number. These program control statements allow you to change the course of execution to suit the processing requirements.

GOTO <line#>

Causes program execution to continue at the line number specified. This is sometimes referred to as an unconditional branch. A LINE# ERROR occurs if the line number is not in the program.

ON <exprn> GOTO <line# list>

Evaluates the numeric expression and converts the result to an integer which selects one position in the <line# list>. Hence, this integer must be from 1 to the length of the <line# list>. Program control is then transferred to the line number selected. The <line# list> consists of a sequence of line numbers, separated from one another by commas, which must already exist in the program. This is commonly referred to as a computed GOTO or a multi-way branch.

STOP, END, DOS

These three statements cause immediate suspension of program execution. STOP puts you back into APCBASIC command level and allows a CONTinuation later on (useful during debugging as breakpoints). END terminates all further execution like STOP, but does not allow CONTinuation. DOS lands you back into the operating system command level, bypassing the APCBASIC command level altogether. Under the runtime version (RUN) all three statements terminate back to the operating system level, as no command level exists in version RUN.

All three statements update any unwritten file data buffers to their respective files prior to actual termination. This is the same as CLOSing all your files except that the files remain OPEN for access upon CONTinuation (STOP only) or by direct statements at the APCBASIC command level.

IF <logical exprn> THEN <statement1> [ELSE <statement2>]

Evaluates the <logical exprn> and if the result is true (non-zero) executes <statement1> and not <statement2>; otherwise (false or zero) <statement1> is skipped and <statement2> is executed when present.

Although the <logical exprn> is usually a simple comparison of some type (eg. IF X=Y THEN...), an expression of any complexity is permitted as long as it produces a numeric result and the entire statement fits on one line. Any combination of string comparisons, numeric comparisions and general numeric expressions is possible (eg. IF A$>B$ OR NOT X+Y AND Z<50 THEN...).

Any single statement except for FOR and NEXT may be used as <statement1> or <statement2>. Since IF..THEN..ELSE is also a single statement, using it here creates a compound or nested IF statement. The optional ELSE clause in such a statement is always associated with the nearest previous IF. For example: IF e1 THEN IF e2 THEN s1 ELSE s2 ELSE IF e3 THEN s3 ELSE s4 would be executed as: IF e1 THEN [IF e2 THEN s1 ELSE s2] ELSE [IF e3 THEN s3 ELSE s4]. The brackets are used only for illustration and need not be employed in actual IF statements. As a special case, if a GOTO <line#> follows a THEN or ELSE clause, the GOTO is optional. For example: IF X>4 THEN 250 ELSE 400.

Compound Statements

For greater expressive power, either <statement1> or <statement2> or both may be a compound statement, which is several ordinary statements grouped together and executed as a unit. To form a compound statement from several individual statements, surround them with brackets []. Compound statements can only appear within IF statements and must fit entirely on a single line, which may extend up to 159 characters (2 full CRT lines). The following example should clarify their use:

```
IF X=Y THEN [R=SQRT(Z+10); SWAP S,T] ELSE [FOR I=1 TO 10; R=R+X(I); NEXT]
```

Notice that FOR..NEXT (and WHILE..NEXT) loops may be included within compound statements. When an IF statement is employed within a compound statement, it too can include compound statements for its THEN or ELSE clauses. You can use the bracketting mechanism to override the normal precedence of ELSE clause processing whenever required, for example:

```
IF X=Y THEN IF A$=B$ THEN S=T ELSE T=S
```

In this example, the ELSE refers to (by default) the 2nd IF which is only executed if X=Y. Suppose that the desired action is to execute the ELSE clause upon failure of the 1st IF test (X=Y). This can clearly be done as follows:

```
IF X=Y THEN [IF A$=B$ THEN S=T] ELSE T=S
```

Of course the complexity of compound IF..THEN..ELSE statement is limited to what you can fit into 159 characters, but this contruct will surely satisfy the vast majority of cases encountered which exceed the possibilities of simple IF..THEN..ELSE statements. Incorrectly formed IF statements result in a SYNTAX ERROR in all but two cases. Examples of each now follow:

```
IF X=Y THEN R=T ELSE T=U ELSE U=V

IF A=B THEN [statements] GOTO 100
```

Too many ELSE clauses are specified in the first example. In such a case, APCBASIC simply ignores the extra ones (ie. they are never executed) and the program continues without reporting the error. The 2nd example illustrates a compound statement followed by another statement (GOTO 100) without a statement separator (; or \) in between. This error is detected as a SYNTAX ERROR if the THEN clause is executed (ie. when A=B). But if it is not executed, neither is the GOTO 100. APCBASIC continues program execution after the end of the IF statement, as defined by a semi-colon or the end of the line. To detect all these cases would significantly degrade IF statement performance beyond the possible inconvenience that not detecting might create. Therefore you should keep these in mind when programming IF statements.

FOR <index variable> = <range1>,<range2>,<range3>,...,<rangeN>

Used in conjunction with the NEXT statement below, the FOR statement provides a general purpose high-speed cycling (looping) method for iterative program structures. Between the FOR statement and its 'closing' NEXT is the main body of the loop, which may consist of any number of program statements (even other FOR..NEXT loops). The idea is to execute a group of statements (located between a FOR and a NEXT statement) repeatedly while setting the index variable to succesive values of each specified range. The ranges are accessed from left to right and may assume one of the following three forms:

(1) <first value> TO <last value> BY <step size>
(2) <first value> TO <last value>
(3) <single value>

Form (1) specifies that the index variable will start at the <first value> and is incremented by the <step size> after each iteration until the <last value> has been exceeded (terminating the loop). Form (2) is the same as form (1) except that the omitted <step size> defaults to a <step size> of 1. Form (3) is the same as form (2) except that the <first value> and the <last value> are the same and exactly one iteration results. All range parameters are specified with general numeric expressions and may evaluate to non-integer values.

These three forms of <range> may be mixed in a single FOR statement whenever necessary. Most of the time you will be using a single <range> of form (2) but look at the following examples for a feeling of the other possibilities:

```
FOR X=1 TO 100          FOR X=175 TO 38 BY -1        FOR X=SQRT(Y) TO Z-10 BY S

FOR X=-12,Y*Z,15,A(F,G)      FOR X=1 TO 10, 20 TO 100 BY 10, 200 TO 1000 BY 100

FOR X=1,2,4,8,10 TO 20 BY 2,-58 TO -1000 BY -7
```

Each iteration begins by comparing the current index variable value with the <last value>. Execution proceeds through the loop body only while the index variable value remains within its defined range. As each <range> is completed, the next <range> is loaded internally and the loop continues. The loop terminates at the end of the last <range> listed in the FOR statement. Zero iterations are possible when the <first value> at the outset exceeds the <last value> (or below on negative <step size>). Since the ranges are evaluated only once and maintained internally, none of the range parameters can be altered during loop execution.

For efficiency, APCBASIC maintains a FOR..NEXT internal structure on the control stack for the lifetime of the loop. Because of this it is imperative that you properly terminate the loop, enabling APCBASIC to remove this internal control structure. Three methods of loop terminations are possible: falling 'out the bottom' when the index value runs out, branching out using the EXIT statement below, or executing a RETURN statement to exit not only the loop but a GOSUB or user-defined function as well (see Section 5.5). Never leave a loop with a GOTO statement! Without proper loop termination, the loop control mechanism will remain on the control stack indefinitely, which can result in a later error at some unpredictable point in the program.

WHILE <logical expression>

Similar to a FOR statement, WHILE implements a general looping structure that repeatly executes a group of statements (terminated by a NEXT statement) until some condition is no longer true. The condition in this case is a logical expression that is evaluated at the start of each iteration of the loop. In order for a WHILE loop to terminate, the body of the loop must at some point cause the logical expression evaluate to zero (false). No iterations through the loop will be made if the logical expression evaluates to zero at the top of the first iteration. For example:

```
WHILE X<100; X=X+1; NEXT        Increments X until it X=100.

WHILE Z=Y; X=X+1; NEXT          Increments X forever.
```

The first example does nothing if X is already 100 or greater. The second example illustrates what happens when the logical expression is not altered the body of the loop. It may be that you desire such an infinite loop in your application because you employ other means to terminate the loop, for example:

```
WHILE Z=Y; X=X+1; IF X>=100 THEN EXIT; NEXT
```

This example illustrates the use of the EXIT statement (described below) in terminating a WHILE loop, bypassing normal termination. The WHILE token in APCBASIC is the same one used in North Star BASIC for the optional LET token (eg. LET X=0). Thus WHILEs found in existing North Star BASIC programs LISTed in APCBASIC must be deleted.

NEXT [<index variable>]

Defines the end of a FOR or WHILE loop. When executed it increments the index variable and re-starts the loop for the next iteration. When supplied, the <index variable> must be exactly the same one defined in the corresponding FOR statement. This is a formality however and is useful only for programming clarity and style (faster loop execution actually results without it). An <index variable> is illegal in NEXT statements that terminate WHILE loops. There must be one and only one NEXT statement associated with each FOR statement and it may not appear immediately after a THEN or ELSE.

EXIT [<line#>]

This special form of GOTO is required to properly transfer program control from an active FOR or WHILE loop before all the iterations have been completed. EXIT with the line number omitted causes control to pass to the statement immediately after the current closing NEXT statement. A line number may be supplied to specify any other EXIT point immediately outside the current loop. EXIT can exit only one level of looping structure and trying anything else results in either an EXIT ERROR or unpredictable program results.

ERRSET [<line# trap> [,<error line variable> [,<error type variable>]]]

When an error occurs during program execution, APCBASIC normally prints an error message and the line number in which it occurred, then terminates execution. ERRSET sets up an 'error trap' which instead re-routes program execution in case of an error. In this manner your program remains in control at all times.

When an error occurs, execution transfers to the line number <line# trap> and sets two simple numeric variables to the offending line number and the error type code (see Section 7.1 for error codes). It also resets the trap until another ERRSET statement is executed setting another error trap (so that errors in your error-processing will not cause infinite looping). Your program can nullify an existing error trap by executing an ERRSET without arguments. An example of each form of ERRSET now follows with brief descriptions.

Sample ERRSET Statement	Operation Performed
ERRSET	Disables the active ERRSET at the current subroutine level until that subroutine returns. No operation is performed if no ERRSET in effect.
ERRSET 125	Sets an error trap so that execution will branch to line 125 if any trappable error should occur. This trap remains in effect until the current subroutine RETURNs or until re-defined by another ERRSET.
ERRSET 125,L	Same as the last form except that variable L will be set to the line number in which the error occured.
ERRSET 125,L,T	Same as the last form except that variable T will be set to the error type code corresponding to the type of error that occured.

The ERRSET statement provides so much freedom that you can get into trouble applying it haphazardly. Each GOSUB or user-defined function may independently set up (or disable) its own ERRSET traps without affecting traps set by higher levels of your program. If a lower level GOSUB or function does not set any traps of its own, any higher level trap will function within the lower level.

Such a transfer out of a lower level to a higher one is fully supported and constitutes the only legal way to bypass the normal RETURN mechanism. For example if GOSUB 100 sets a trap then calls GOSUB 200 which in turn generates an error, the original trap is used. However if GOSUB 200 sets its own trap before the error was encountered then that trap is used. Upon RETURN, GOSUB 100 is still protected by its original trap, unaffected by any ERRSETs within GOSUB 200.

What this means for applying ERRSETs is that the error trap line number should be a line at the same program level as the ERRSET statement that assigns it. Specifically, never assign an error branch that can jump out of the current GOSUB, user-defined function or FOR..NEXT loop that contains the ERRSET itself. Instead, assign a trap to a line within the same structure. For example, to transfer control out of a FOR..NEXT loop when an error occurs within the loop, execute the ERRSET statement prior to entering the loop, with an error trap referring to a line also outside the loop (at the same level as the ERRSET statement). This is actually the most straight forward and easily debugged method for constructing error traps within a procedure oriented language of any type. APCBASIC cannot enforce these rules when the ERRSETs are made, so you must be careful to apply them properly.

Local variables with GOSUBs or FNs will never be restored to their original state if an error trap transfers out of that subprogram to a higher level. In fact, all data localized between the error level and the trap level will be in effect at the trap level and their restored values will be unrecoverable. You must be aware of this shortcoming of APCBASIC error processing in order to deal with this issue as needed.

ERRSET #<numeric error type exprn>

Generates an error of the given type for purposes of debugging and special program control applications. You must have an active ERRSET in effect at the time this statement is executed, otherwise a USER TRAP ERROR is issued. For example: ERRSET #7 generates a FILE ERROR exactly the same way that OPENing a non-existent file would. The <type> can be a numeric expression that evaluates to a value between 0 and 127.

5.4

-- 5.5 SUBROUTINE STATEMENTS --

A subroutine is a procedure that includes apparatus permitting its use by simply referring to it, instead of repeating the same section every time it is needed. This allows efficient use of memory and provides a means for 'hiding' the details of such procedures to clarify program logic. Programs are generally built from subroutine building blocks. Subroutines themselves may be constructed from other subroutines at 'lower levels', and so on.

APCBASIC provides two types of subroutines: GOSUBs and User-Defined Functions. The GOSUB is simply a means for re-using a section of program lines from any place in the program. User-Defined Functions have in addition the ability to pass argument data to the function procedure through local variables and return result data back to the requesting entity. The bulk of the discussion on such functions is found in Sections 3.6 and 4.5 and only a brief syntax description is provided below.

GOSUB <line#>

Short for GOTO Subroutine, a GOSUB statement transfers program control to the line number specified as with the GOTO statement. However in a GOSUB, APCBASIC keeps track of the statement following the originating GOSUB so that when a RETURN statement is executed control returns to it. Although the GOSUB (referred to as a GOSUB call) jumps to a specific line number, there may be any number of RETURN statements within the body of the GOSUB subroutine, and any of them will RETURN the same continuation statement when executed.

Think of GOSUBs as program blocks that perform a procedure as an operational unit. Although the body of a GOSUB has no obvious structure required by APCBASIC language syntax, it is important to treat it as a unit by clearly defining its entry and exit points and using them in rigidly controlled ways. APCBASIC provides several mechanisms that depend on well-defined block structured GOSUBs to be useful. Error processing structures (ERRSETs) and DATA READ pointers are local within GOSUBs and within User-defined functions. This means that changes made to these areas do not affect or propagate back up through to the program when the GOSUB returns. You can define your own LOCAL variables within GOSUBs which may be used in any way whatsoever without affecting anything outside that GOSUB (discussed below). Awareness of these features is necessary for proper programming of GOSUBs, DATA-READs and ERRSET processing.

RETURN [<exprn>]

Directs program control to the statement following the most recent GOSUB call, or returns a string or numeric expression result from a user-defined function. The <exprn> is always and only used in RETURNs from User-Defined Functions (described in Sections 3.5 and 4.6).

Before the actual return, RETURN restores the state of the previous READ pointer (for DATA statements), the previous ERRSET structure and any LOCAL variables (see LOCAL statement below) to their state at the time of the GOSUB or user function call.

ON <exprn> GOSUB <line# list>

Evaluates the numeric expression and truncates the result to an integer that specifies a position in the <line# list>. This integer must be from 1 to the length of the <line# list>. Program control transfers to the GOSUB selected as in the ON..GOTO statement of Section 5.4. This is commonly referred to as a computed GOSUB. RETURNs pass control to the statement following the <line# list>.

LOCAL <list of string & scalar variables>

Creates temporary simple string and numeric variables (not arrays) which may be used freely for any purpose within GOSUBs and FNs. Global variables of the same name which already exist are protected but inaccessable until the sub-program executes a RETURN statement. Scalar & string variables may be listed separated by commas after the LOCAL keyword, and can then be used for unlimited local working storage. Since local variables carry their previous value after the LOCAL declaration, they may be employed for passing data parameters to GOSUBs. On RETURN, their prior values are restored, and program execution resumes. Use within recursive procedures to create temporary working variables makes this is a particularly useful and powerful tool. See the discussion of recursive programming at the end of Section 3.5.

Subscripted variables and strings exceeding 255 characters cannot be LOCALized. Also, LOCAL declarations are illegal if not within a GOSUB or FN, or if directly within FOR or WHILE loops. Re-DIMensioning of a LOCAL string is permitted as long as its previous string value will fit upon RETURN. The new DIMension will remain in effect after the current GOSUB RETURNs.

LOCAL variables will not be restored to their original state if you branch out of a GOSUB or FN using them with a GOTO, instead of a normal RETURN. This is also the case when the branch is an error trap to a higher program level.

DEF <function name>[(<argument list>)][=<expression>]

Defines a User-Defined Function, including its name, its list of parameters and its mode of operation: single or multiple line DEFinition. The parameter list, if it exists, consists of a sequence of unindexed string or numeric variable names, enclosed in parentheses as shown. If the <expression> at the end of the DEF statement is omitted, then a multiple-line Function is DEFined and whose procedural DEFinition must follow. This consists of a sequence of statements that includes at least one RETURN <exprn> statement and ends with an FNEND statement (described below). The DEF statement must appear as the first statement on the line in which it appears. Section 3.5 contains further details on User-Defined Functions.

FNEND

Used as the last statement of a multiple-line User Function to indicate where its DEFinition ends. Unlike the DEF statement, which must be the first statement on a line, the FNEND statement may appear anywhere on a line as long as it is the last statement of the Function. Single-line User Functions do not use the FNEND statement.

-- 5.6 FILE PROCESSING STATEMENTS --

Described below are the file statements from the North Star DOS perspective. See the CP/M exceptions to these in Section 7.4 and be sure to see 6.4 for various file functions: FILE(), FILEPOS(), FILESIZE(), TYP() and SPACE(). A file is identified with a sequence of characters known as its file name. Consult your operating system manual for details concerning file names, file types, and internal file structures.

Files may be CREATEd, DESTROYed, RENAMEd, OPENed and CLOSEd. An OPEN file can then be READ or WRITten in mixed sequential and random access modes. APCBASIC provides random access to the byte level, with file capacities to 16 megabytes (8 megabytes under CP/M). The CP/M version also fully supports dynamic files. Data formats supported are Floating Point values, Proper Strings, Binary Strings, 8-bit values and 16-bit values.

CREATE <new file name>,<file size>[,<file type>]

Creates a new file on the disk. The <new file name> must not already exist and will become a type 3 file unless the <file type> option requests a different file type number (0-127). The <file size> is the number of file blocks to allocate to the new file (256 byte increments). Omitting a drive reference from the file name refers to the default drive number (usually 1). The file name is a string expression; size and type are numeric expressions.

DESTROY <existing file name>

Permanently deletes the specified file name from the disk and its directory. A FILE ERROR results from specifying a non-existent file name. The file name is given as a string expression.

RENAME <old file name>,<new file name>

Programs can RENAME files using this statement. APCBASIC reports a FILE ERROR if either the <oldname> doesn't exist or the <newname> does exists. Both file names are given as string expressions.

OPEN #<file number> [% <file type>], <file name> [, <size variable>]

Associates a file number (0-15) with an existing file to provide program access to that file in either sequential or random access mode. The file type is assumed to be type 3 unless otherwise specified by the optional <file type> expression. A size variable may be supplied to receive the number of file blocks in the file at OPEN time. This provision maintains compatibility with North Star BASIC but is unnecessary since the FILESIZE() function in Section 6.4 does the same thing in a more obvious way.

OPEN assigns each file a 512-byte internal buffer that acts as a high-speed interface between APCBASIC and the disk operating system. A file may be OPENed under several different file numbers simultaneously. This can be most useful when several independent file buffers are indicated, as in sorting programs. APCBASIC controls multiple buffer usage so that at all times any given 512-byte file segment can only be buffered by at most one buffer. This is transparent to the program and prevents any file update problems due to the multiple buffers.

Files OPENed under file numbers 8-15 may be sequentially accessed by PRINT or INPUT statments, in addition to by READ and WRITE statements. This facility provides efficient sequential access to text-files. See Section 5.2 for the details (under PRINT and INPUT).

CLOSE [#<file number>]

Disassociates an OPEN file from its APCBASIC file number and writes any current data remaining in its file buffer to the file. The <file number> specifies the file number (0-15) to CLOSE. Omitting it CLOSEs all OPEN files. The file buffer remains in memory until reassigned to the next newly OPENed file or until released to free memory by the FREE statement discussed below.

FREE

The FREE statement (not to be confused with the FREE() function) will release all dead buffer space left by files after they are CLOSEd. This statement must NEVER be used while any FOR..NEXT loop or FN is in progress as it may cause a APCBASIC system failure. No arguments are specified with this statement. Memory released is then available for use in new variables and internal working storage.

READ #<file number>,<data variable list>

READs data from a file in sequential or random mode from the previously OPENed file number specified. The data variable list consists of numeric variables, string variables and file position specifications, in any combination and separated by commas. If positioning is omitted, all data is automatically READ sequentially from ascending file positions.

File positions are resolved to the byte level (numbered from zero) and are specified as numeric expressions preceded by a percent sign (%) to indicate the special positioning function. Several random file position changes may be specified within the same READ statement data list if desired. Example: READ #F,%P1,X,Y,%P2,Z reads X and Y from position P1, and Z from position P2.

Simple numeric variables and individual array elements may be READ from the file. Therefore, the numeric precision of APCBASIC must match the precision in effect when the data was written. Binary data can also be READ into numeric variables by prefixing each numeric variable reference with an ampersand (&) for 8-bit values or an at-sign (@) for 16-bit values. (Note that 16-bit values are defined as the next two bytes on the file with the low-order byte first.) Binary file operations bypass all type checking since they READ whatever is presented to them. The 8-bit and 16-bit values are converted to floating point format when READ from the file into numeric variables. For example, READ #F,X,A(I,J),&Y,@B(K) illustrates the various numeric READ methods.

String variables can be READ as a whole or as indexed sub-strings. String data is written in a special compact format: the amount of storage taken equals the length of the string plus two for up to 255 characters, or plus three for over 255 characters. Regardless of the string variable capacity, the file pointer is always set properly to the next item in the file after READing any string. Preceeding the string variable name with an ampersand (&) READs 8-bit binary data directly into the string, with the number of bytes READ controlled by the current (before the READ) length of the string. 16-bit binary READs are not possible with string variables. For example READ #F,A$,B$(I,J),&C$,&D$(K,L) illustrate the string READ possibilities.

As each data item is READ from the file, the file position is incremented by the number of bytes READ, so that the file position is always aligned to the next data item. When randomly accessing a data file, you must specify file positions which always refer to the 1st byte of multi-byte data items (such as strings, 16-bit values, and floating point values. To do this you must know the number of bytes required for each data item. String and binary data types are covered above. The length of floating point values is always the same for a given APCBASIC precision: PRECISION/2+1. Thus the standard 8-digit precision requires 5 bytes (8/2+1 = 4+1 = 5). If you ever access a data item somewhere past its 1st byte, a TYPE ERROR will usually occur to inform of the problem. However binary data items have no identifying characteristics to permit such error detection, so exercise great care when processing random binary files.

WRITE #<file number>,<data exprn list> [,NOMARK]

WRITEs a data list to the specified OPEN file in either sequential or random access mode. Like the READ statement except for the direction of data transfer. Instead of variables in the data list, general data expressions can specify the data to be written. The data expression list may contain numeric expressions, string expressions and file position specifications. See the previous discussion on the READ statement for details on file positioning and numeric data on files. Example: WRITE #F,%P1,X,Y,%P2,Z writes X and Y at position P1, and Z at position P2.

Ordinary numeric expressions specify the floating point data to WRITE. Binary data can also be written by prefixing each numeric expression with an ampersand (&) for 8-bit values or an at-sign (@) for 16-bit values (see the READ statement for details). Floating point values from 0 to 65535 are converted to binary format before the WRITE takes place. For example, WRITE #F,X,A(I,J),&Y,@B(K) illustrates the various numeric WRITE possibilities.

String expressions may also be included in the data expression list and are WRITten in a manner corresponding to the READing of strings. Preceding string expressions with an ampersand (&) WRITEs the string as a sequence of binary bytes the length of the string. No type or length information is WRITten as in the usual string WRITE operation described above. For example, WRITE A$,B$(I,J),&C$,&D$(K,L) illustrates the string WRITE possibilities.

After executing each WRITE statement, APCBASIC WRITEs an additional single-byte end-of-file mark at the prevailing file position (without advancing the file pointer however). While useful for purely sequential file usage, this often proves unsatisfactory for binary or random access operations. To prevent the generation of the end-of-file mark, you may finish WRITEs statement with the NOMARK keyword. This same keyword may also be used as a program statement to provide global control over the generation of end-marks for subsequent WRITE operations (see the next statement).

NOMARK <logical exprn>

The end-of-file mark written after each WRITE operation may be suppressed (or not) for all subsequent WRITE operations by using the NOMARK statement. A non-zero expression (logical true) causes file mark suppression, and a zero (logical false) expression brings it back again. Most useful for random or binary file processing, since it eliminates the need for the NOMARK keyword in WRITE statements. The NOMARK statement affects the action of only those WRITE statements which omit the NOMARK keyword; any WRITE statement terminated with NOMARK will never WRITE and end-of-file mark.

DIR [#<device number>,] [<drive number>]

To generate a file directory listing from a APCBASIC program, the DIR statement may be executed from a APCBASIC program in the same manner as the DIR command described in Section 2.4.

— 5.7 SEGMENTATION & OVERLAY STATEMENTS —

Accepting the reality of finite memory, very large programs must be partitioned into components that reside on disk files which are called into memory as needed. Three APCBASIC statements provide the ability to automatically LOAD & RUN a program, MERGE subroutine libraries into the currently executing program, or DELETE arbitrary program lines no longer needed by the current executing program.

Use of MERGE or DELETE requires that you keep a ´map´ of your program line assignments both in mind and well documented, or you can introduce strange errors of logical integrity into your programs by DELETing or overlaying the wrong lines. Both MERGE and DELETE should therefore be used with great care.

LINK <program name string exprn> [,<common variables>]

Terminates the current program, erases it, loads another program specified by the string expression, then begins execution on the first new program statement. All files are generally closed and data stored in variables may or may not be lost, depending on the <common variables> portion of the LINK statement. LINK thus provides a means for APCBASIC programs to automatically LOAD & RUN program segments of their own choosing. Unless you have TRACE on, LINK removes all spaces and REMarks from the new program just prior to RUNning it.

Variables may be passed between LINKed programs by listing their names after the program file name expression in the LINK statement. Any type or size of variable may be passed as long as space in the LINKed program permits. For example the statement: LINK "PGM",X,Y,B$,V() will LINK to PGM and pass X,Y,B$,V() to it, where () indicates that V is an array. Syntax errors result from specifying subscript expressions or unused variables.

To preserve all variables, use an at-sign (@) instead of the variable list. For example LINK "PGM",@ will pass all variables to PGM. With this method, any files OPEN before LINKing will still be OPEN when the LINKed program begins. At-sign LINKing permits writing programs as though the machine possessed unlimited memory for the program statements.

MERGE <program name stringexprn>

Program files may be MERGEd during program execution in the same way as described in Section 2.2. Sophisticated overlay structures can be created using this feature. For instance, your ´core´ routines can always stay in memory while special purpose libraries can be MERGEd into the program as they are needed. Use the same syntax as the MERGE command, but with the program name expressed as a string expression.

Execution continues at the 1st line of the program after a MERGE and some method must be employed to restart at your desired point of continuation. Since all variables retain there values (and open files remain open), the 1st program line may branch on a previously defined control variable using an IF..THEN or ON..GOTO statement. Another method is to MERGE a new line 0000 into the program that jumps to the desired continuation line.

The result is essentially a new program. You must make sure that all line references are resolved and that DEFined FNs are not duplicated. In general the program must be intact and complete after each MERGE executed. Because the CRUNCH utility program (Section 8.2) shifts many program statements to adjacent lines (removing their line numbers in the process), you must take this into account when you set up your MERGE operation to guarantee correct MERGing of CRUNCHed programs.

DELETE <line# range>

To remove unneeded program lines after they have been executed (such as one-time initialization routines), use the DEL statement in the running program just like the DEL command. All variables and open files are preserved. Like the MERGE above, control passes to the 1st line of the program after executing this statement. The only restriction is that the DEL statement must appear within the range being deleted. You must guarantee that the resulting program is complete in all respects as with the MERGE statement.

When you compact a program containing DELETE statements using CRUNCH (Section 8.2), be sure that the line immediately following each DELETEd range is referenced elsewhere in the program (eg. by GOTOs, GOSUBs, ERRSETs, etc). CRUNCH joins unreferenced lines with its neighbors wherever possible (while preserving all references), and could otherwise append additional statements to each of the specified DELETE ranges.

— 5.8 SYSTEM INTERFACE STATEMENTS —

These statements provide access to system memory and hardware ports 0 to 255 and permit control of various APCBASIC system parameters. See Section 6.5 for the discussion of additional functions: FREE(), EXAM(), INP(), CALL(...) and variable addressing.

FILL <starting address>,<data list>

Stores a list of data values directly into sequential memory locations. Numeric values are reduced modulo 256 (8-bit value) unless otherwise specified by placing an at-sign (@) in front of a FILL-value, in which case a 16-bit value is stored instead. (16-bit values use 2-bytes in which the 1st & 2nd bytes are the low & high order bytes respectively.) String expressions may also be specified for FILLing into memory. The number of bytes stored depends on the length of the string expression.

EXAM <starting address>,<variable list>

Loads string or numeric variables directly from memory and is the inverse operation of FILL. The first variable is loaded from memory at the address specified, and subsequent variables are loaded sequentially from memory in the order given. By placing an at-sign (@) in front of an EXAM-variable, a 16-bit value is loaded from memory instead of an 8-bit value. String variables may also be EXAMined from memory. The number of bytes loaded depends on the current length of the string in the variable.

OUT <port number>,<8-bit data value>

Sends an 8-bit value (0..255) out through the hardware port specified. No status interrogation is performed and the transfer takes place immediately. The OUT statement will accept either numeric or string data for output through CPU ports. For example: OUT P,C$ will output the 1st character in string variable C$. Any general string (or numeric) expression may be specified, however only the 1st character of the string is OUTput. If a null string is specified, an undefined value is OUTput.

CALL <address exprn>,<data register exprn$>[,<result register vbl$>]

Executes a machine CALL to the (16-bit) memory address specified by the numeric <address exprn>. This statement permits machine register access on both the call (input registers) and the return (result registers). Register values are specified as characters in the string arguments and are positionally defined: ACC, F, B, C, D, E, H, L. The data registers may be a string expression of any length. The result register string vbl must be at least 8 bytes long to hold all returned register values (when this optional parameter is supplied). Use BIT(), ASC(), CHR$(), FILL and EXAM to pack/unpack your desired values to/from the string arguments.

PARAM(<exprn>) = <exprn>

The PARAM(P) statement allows control of several internal execution factors. It may be used on the left side of an assignment statement (=) to assign new values, or accessed as a function to determine current PARAM() values. Expression P must evaluate to a value from 0 to 8 to select one of the following parameters:

0 Version number of the current APCBASIC release.

1* CTRL-C Disable may be set to a non-zero value to ignore a CTRL-C typed during progrm execution, or to zero to re-enable the CTRL-C apparatus. This has no effect on CTRL-C typed at the command level or during direct statement execution.

2* Default Drive Number (Initially 1) used whenever drive #0 or no drive is specified in commands, statements, functions and file names.

3* Default I/O Device (Initially 0) used whenever an optional device number is omitted. Has no effect on the console messages displayed by APCBASIC (Ready, error messages, etc).

4 Prevailing Numeric Precision (Returns 8, 10, 12, or 14)

5 Operating System Environment Code. 0=North Star DOS, 1=CP/M, 2=APC's MTOS.

6 Address of original invoking (system level) command that was typed to auto-run the program. Returns 0 if none. Use with EXAM statement to access the command tail, which ends with a carriage return (ASCII 13) when present.

7* Specifies the ASCII code to be used in initializing string variable on creation (or restoration). PARAM(7)=32 on start-up (for spaces).

8 Addresses the first memory byte above APCBASIC that is not reserved or in use by APCBASIC. Thus data placed their is accessible by your programs and is unaltered by by multi-segement execution. Since APCBASIC normally assigns all available memory to your program and its data, no memory space exists at PARAM(8) until you assign lower memory bounds with the CONFIG utility program described in Section 8.3.

All may be 'read' but only those marked (*) may be altered. Expect further parameter definitions in later releases. See Section 6.5 for further details.

-- 5.9 DOCUMENTATION STATEMENTS --

REM <descriptive text>

Everything from the REM keyword to the end of its program line is taken as a non-executing comment, including statement separators (; and \) or text which would ordinarily constitute valid executable statements. APCBASIC preserves the case (upper/lower) of all letters that follow the REM keyword.

REM statements provide additional information and guidance to the programmer during program development and later program maintenance. Well commented programs generally take less total time to construct and debug. A good practice to adhere to is to briefly describe each procedure or subprogram in its first line. Also, type a line-feed as the last character of each program line that precedes such REMarks. Since this creates double spacing between procedural blocks of code, the program structure becomes much more evident.

5.9

**** 6.0 APCBASIC FUNCTION LIBRARY ****

Unless otherwise noted, all parameters to functions can be general string (denoted by S$, T$ or U$) or numeric expressions (denoted by X, Y or Z).

-- 6.1 ARITHMETIC FUNCTIONS --

INT(X) Returns the greatest integer less than or equal to expression X. Examples: INT(34.524509)=34, INT(.2)=0, INT(-7)=-7, INT(-1.3)=-2

CEIL(X) Returns the lowest integer greater than or equal to expression X. Examples: CEIL(3.4)=4, CEIL(-2.13)=-2, CEIL(1)=1

TRUNC(X) Returns X with any fractional part removed and is equivalent to ABS(INT(X))*SGN(X). Examples: TRUNC(3.4)=3, TRUNC(-2.71)=-2

MOD(X,Y) Returns the smallest non-negative remainder R such that X-R is exactly divisible by Y. Examples: MOD(34,17)=0, MOD(13,5)=3, MOD(-13,5)=2

FRAC(X) Returns the non-negative fractional part of expression X and is equivalent to X-INT(X). Examples: FRAC(3)=0, FRAC(4.23)=.23, FRAC(-7.2)=.8

ROUND(X) Returns X rounded to the nearest integer. Note that this is easily generalized to round to the nearest value modulo V by using the expression: V*ROUND(X/V).

ROUND(X,P) Returns X rounded to the number of significant digits precision specified by expression P, which must be 1 or greater.

ABS(X) Returns the positive value of X. Examples: ABS(-3)=3, ABS(2)=2

SGN(X) Returns -1, 0, or 1 for X<0, X=0, or X>0 respectively. Examples: SGN(-4.5)=-1, SGN(0)=0, SGN(3521)=1

SGN(X,Y) Returns Y with the sign of X, irregardless of the sign of Y. Examples: SGN(-4,10)=-10, SGN(0,-34)=34, SGN(-3,-99)=-99

MIN(X,Y,..) Returns the minimum value among a list of expression values. Example: MIN(45,2,987,-12,0,34)=-12, MIN(2,1)=1

MAX(X,Y,..) Returns the maximum value among a list of expression values. Examples: MAX(45,2,987,-12,0,34)=987, MAX(2,1)=2

INDEX After a MIN(..) or MAX(..) function call this system variable returns the position of the value returned. After the MIN(..) examples above, INDEX returns 4 and 2 respectively; after the MAX(..) examples it returns 3 and 1 respectively.

RND(X) Returns a pseudo-random number sequence uniformly distributed over the interval 0...1, not inclusive. The expression value X controls the method of computation: X=0 causes the return of the next number in the current sequence; 0<X<1 defines a new starting 'seed' which is derived from X; and X<0 defines a new starting seed based on a random hardware condition.

-- 6.2 MATHEMATICAL FUNCTIONS --

SQRT(X) Returns the square-root of the non-negative expression X. Examples: SQRT(9)=3, SQRT(1)=1, SQRT(5.7)=2.38746727...

LOG(X) Returns the logarithm base 10 of any X>0. Example LOG(1000)=3

LN(X) Returns the logarithm base e of any X>0.

EXP(X) Returns e raised to the power of X. To avoid a numeric overflow error, X must be within a range of -147.36549 to +145.06286.

PI Returns the constant pi (3.141592..) rounded to the prevailing precision of APCBASIC.

SIN(X) Returns the sine of X specified in radians. Note that X must be multiplied by PI/180 if it represents degrees.

ASIN(X) Returns the angle in radians of X specified as a sine from -1 to 1. In other words, if X=SIN(A) then ASIN(X)=A.

COS(X) Returns the cosine of X specified in radians.

ACOS(X) Returns the angle in radians of X specified as a cosine from -1 to 1. Thus if X=COS(A) then ACOS(X)=A.

TAN(X) Returns the tangent of X specified in radians.

ATN(X) Returns the angle in radians of X specified as a tangent. Thus if X=tangent(A) then ATN(X)=A.

POLY(X,A(),D) Returns the polynomial evaluation of X using coefficient array A() containing D+1 coefficients (D=polynomial degree). Note that the array A() specifies the starting coefficient (constant term) of an ascending coefficient sequence stored in A(). D must be a value from 1 to 255. Invalid evaluations result if your degree and coefficient sequence extends past the end of the array.

Multi-dimensional arrays may be employed with the understanding that their last dimension index specifies the coefficient sequence position, and prior dimension subscripts serve to select one sequence of many. This follows from the sequential coefficient access employed by POLY() and the organization of array storage elements. For example POLY(X,C(I,J),5) evaluates a 5th-degree polynomial using the coefficient list C(I,J) on X, where I selects the sequence and J specifies the low-coefficient position of the six coefficient sequence (coefficient 0 to 5).

LEN(S$) Returns the length of the string expression S$. This function requires more execution time and memory space when the string is not a simple string variable or constant.
Examples: LEN("ABCDEFG")=7, LEN("")=0

STR$(X) Returns the ASCII string representation of the value of X according to the current default numeric format.
Examples: STR$(123.70)=" 123.7", STR$(-34E2)=" -3400"

STR$(X,S$) Same as STR$(X) except that a format string is given to control the format of the value. Examples: STR$(23.87,"6F1")=" 23.9", STR$(1234567.8,"CI")=" 1,234,568"

VAL(S$) Returns the numeric value of an ASCII string representation of a value (the inverse of the STR$(X) function). The string may contain leading or trailing spaces (and/or line-feeds), but must be an otherwise valid numeric constant.
Examples: VAL(" 92E3")=92000, VAL("0012.4300 ")=12.43

CHR$(X) Returns a one-character string corresponding to the ASCII code given by X. Examples: CHR$(65)="A", CHR$(57)="9"

CHR$(X,Y) Returns a string of the ASCII codes X through Y in ascending sequence. IF X>Y then a null string is returned. Examples: CHR$(48,57)="0123456789", CHR$(87,43)=""

ASC(S$) Returns the ASCII code value corresponding to the 1st character in string S$. Examples: ASC("A")=65, ASC("9")=57, ASC("")=-1
S$ may be a general string expression, but only the 1st character is used for the result. ASC(S$) executes faster and uses less memory when S$ is a simple variable.

TRIM$(S$) Returns S$ stripped of all leading and trailing spaces.

REV$(S$) Returns S$ with the characters in the reverse order.

TRAN$(S$,T$,U$) Returns S$ after substituting (translating) any characters also contained in T$ to corresponding characters in U$. When T$ and U$ differ in length, the longer is truncated to length of the shorter. If T$ or U$ is null ("") then S$ is returned unchanged.
Example: TRAN$("ABCDEFG","BDFR","----")="A-C-E-G"

Internally, TRAN$() requires an additional workspace area and thus erases the prior content of the 'old line' editing buffer. Neither replacement string (2nd and 3rd arguments) may exceed 256 characters without causing an OUT OF BOUNDS ERROR.

MATCH(S$,T$) Searches string S$ for the 1st occurrence of string T$ and returns the character position in S$ where the first character of T$ was found; or returns zero if no exact match was found.
Examples: MATCH("abcdefg","de")=4, MATCH("abcdefg","DE")=0

MATCH(S$,T$,X) Same as the MATCH function above except the search process begins at position X in S$ instead of the first character position. Examples: MATCH("abcdefg","de",5)=0

FIND(T$=S$,X) Searches string variable T$ for the first occurance of string expression S$ (which must not exceed 255 characters) that satisfies the relation specified by the comparison operator (in this case '='). Any comparision operator may be employed (=, <>, <, >, <=, or >=). The last argument is optional and specifies the number of positions to advance the search for each successive comparison. For example, X=5 compares S$ with T$ at positions 1,6,11,16,21,... until the relation is satisfied. Omitting this argument implies a default of 1. To search T$ backwards from the end to the beginning, specify a negative X. In such a case, the search begins from the last possible comparison position in T$ (ie. LEN(T$)-LEN(S$)+1).

To control the extent of the search, T$ may be indexed to the desired region. FIND returns the position in T$ that satisfied the relation, or it returns zero if not satisfied. This position is always relative to the beginning of the string variable, no matter how you index T$. For convenience, FIND sets INDEX to the result position relative to the indexed region in T$ actually searched (so that it is available for subsequent use when needed).

You may have noticed the similarity bewteen FIND and MATCH. In fact, FIND(T$=S$) returns the same result as MATCH(T$,S$). However FIND has several differences that should be emphasized. FIND can search string variables containing huge strings, but since MATCH requires working memory to hold all of its parameters, it fails if memory fills up (like searching a 20000 byte table). FIND is designed to find relationships in string tables by searching forward or backward through fixed length sub-strings. MATCH is designed for simple pattern matching in relatively small strings.

BIT(V$,X) Returns the value of the Xth bit of string variable V$. X must evaluate to a value from 0 to LEN(V$)*8-1. V$ may be subscripted or unsubscripted, but cannot be a string expression. Only values 0 or 1 are returned. Use on the left side of an assignment statement to store values and on the right to obtain values. See the BIT() statement described in Section 5.3 for further details.

BIT(V$,X:Y) Same as the BIT function above except that Y-bits are accessed as one value. Y may be a value from 1 to 16. The range of values accessed depends on Y. For example if Y=4, then four-bit numbers ranging 0-15 are possible; if Y=13 then 13-bit numbers ranging 0 to 8191 are possible.

ROTAT$(S$,X) Rotates the bit-string S$ by the number of bit-positions specified by X. Negative X rotates S$ to the left (toward the beginning) and positive X rotates to the right (toward the end), consistent with BIT() addressing described in Section 5.3. No action is taken if either S$="" or X=0. X may take on values from -255 to 255. Note that the entire string rotates as a unit and bits that 'fall off' the end are moved to the other end (ie. no information is lost).

-- 6.4 FILE AND DEVICE I/O FUNCTIONS --

POS(X) Returns the current column position of device 0-15 given by X. The result ranges from 0 to 255.

LINES(X) Returns the current line position of device 0-15 given by X. The result ranges from 0 to 255 and may be initialized to zero at any time with PRINT statement containing a plus sign (+) control. Examples: LINES(0)=124; PRINT +,; LINES(0)=0

INCHR$(X) Returns a one-character string containing the next data byte input on device X, where X must be from 0 to 15. It will wait as long as necessary until a character is available from the input device specified. This method of input does not use the carriage return terminator (like the INPUT statement). Control characters and carriage returns may be input as ordinary characters using INCHR$.

You can read console (keyboard) characters without waiting, by specifying any negative value for the device number D in INCHR$(D). INCHR$(-1) always returns a one-character string, even if no console character was typed. So you must test the result character for the no-character-typed code. CP/M versions return an ASCII 0, but other operating system system conventions are different and this code, although consistent, must be determined specifically in your operating environment (eg. under North Star DOS). You must disable the CTRL-C detection system to use this feature (PARAM(1)=1 would do it) because it immediately swallows any character appearing from the console.

INCHR$(X) will now input single characters from a file if the file is OPEN under file number X, where X ranges 8 to 15. This was added to be consistent with the other I/O redirection facilities in APCBASIC. Exactly one (1) character is read with each reference to INCHR$(X). No end-of-file mark is checked for, although an OUT OF BOUNDS ERROR will occur if you attempt to read past the physical end of the file.

Do not use INCHR$(X) in this manner directly in the data list of a WRITE statement to the same file, as it will upset the file pointer for the subsequent WRITE operation.

FILE(S$) Returns the file type of a file specified in S$. If non-existent file name specified then it returns -1. Under CP/M, FILE() returns 1 if the file exists and 0 if it does not exist.

FILEPOS(X) Returns the position of the file pointer of open file number X.

FILESIZE(X) Returns the number of file blocks in the open file number X.

SPACE(X) Returns the number of file blocks (256 bytes) available on the disk drive given by X. X=0 for the default drive, X=1,2,3,... to refer to any attached disk system.

TYP(X) Returns the data type at the current file position of file number X. The data types returned are as follows: 0=ENDMARK, 1=String, 2=Floating Point Value, 3=Unknown (most likely binary data).

-- 6.5 FUNCTIONS FOR SYSTEM INTERFACE --

PARAM(X) Returns an internal APCBASIC condition selected by X, where X may take on values from 0 to 8. Each PARAM is explained below. Certain PARAMs may be set with an assignment statement (eg. PARAM(1)=0). Such PARAMs are marked with an asterisk (*).

PARAM	D e s c r i p t i o n
0	Returns the APCBASIC version number, taking the form: U.VWX, where U=year digit, V=sequence number in that year, W=type of arithmetic processing used (0=software, 1=N* floating point board), X=0 for development version, X=1 for runtime version. Other indicators may be appended to this form from time to time.
1*	CTRL-C disable flag (1=disabled, 0=enabled).
2*	Default Disk Drive number (normally 1).
3*	Default I/O Device number (normally 0).
4	Prevailing Floating Point Precision (8-14)
5	Returns a code specifying the operating system environment under which APCBASIC is executing (0 for North Star DOS, 1 for CP/M, 2 for APC's MTOS).
6	Returns the address to the original command line typed that loaded APCBASIC and a program for immediate execution. Zero is returned if no auto-run program was specified. Use this feature to pass additional parameters to your program from the calling command sequence with the statement: EXAM PARAM(6),L$. Your program must perform all necessary decoding; the line always ends with a carriage return code (ASCII 13). Since some systems (eg. CP/M) reuse the input area for other things, be sure that your program reads this parameter line at the start of program execution.
7*	Permits access to the ASCII code used to initialize strings and string arrays. At startup, PARAM(7) = 32, the ASCII code for spaces (blanks). You can revise this value to any code from 0 to 255 with an assignment statement: PARAM(7)=0.
8	Returns the address of the 1st memory location above memory used by APCBASIC. Your APCBASIC can be made (using the CONFIG program) to leave some memory unused at the top for use by external machine code or data. PARAM(8) will always tell you where this is located.

FREE(X) Returns the number of memory bytes available at any point in time given positive X; returns the starting memory address of that free space given a negative X. Watch out! This starting address is where the next new variable or file buffer will be created and it will overwrite anything you place there.

EXAM(X) Returns the value of memory byte at the address specified by X. Note the distinction between this and the EXAM statement described in Section 5.8.

INP(X) Returns the value of an input data byte from hardward port X. X is reduced to modulo 256 when X>256. No status is examined and whatever data byte is present is returned immediately.

Higher processing speed is possible by employing INP(X) as a string function so that it can return a single character string (which avoids a floating point conversion). Used where a string is expected, INP(X) returns a 1 byte string. Otherwise, a numeric (floating point) result is returned.

CALL(X,Y,..) Returns the value left in CPU register HL after return from a machine subroutine at memory address X. One or more arguments may be supplied. With two or more arguments, the last is passed through register DE and those between the 1st and the last are placed on the hardware stack in reverse order. Each argument is converted to a positive 16-bit integer before passing it. The machine subroutine is responsible for removing the proper number of arguments from the stack, which could not otherwise return properly. See the CALL statement in Section 5.8 for a more general CALL that permits full access to all machine registers.

[V] Returns the memory address of variable V, which may be a string or numeric scalar or array element. String addresses refer to the first character of given string or indexed string. Numeric addresses refer to the exponent byte of the value, which is the last byte of the number. Can be used in CALL functions for passing pointers to data to be processed, or in FILLing or EXAMining their memory contents directly from your program.

The addresses of APCBASIC variables may change during program execution after certain operations: FREE statement, DIM, MERGE, DELETE, and LINKs that perserve variables. Therefore external processes using such pointers must not assume static locations unless you exercise special care.

**** 7.0 MISCELLANEOUS INFORMATION ****

This section covers various subjects that are less likely to be needed on a regular basis, but are nonetheless of vital importance at certain times. The following areas are covered:

7.1 Error Messages	Descriptions of causal factors that can lead to the generation of both trappable and non-trappable errors.
7.2 Alternate Keywords	Lists various keywords that may be substituted for primary keywords for purposes of brevity and convenience.
7.3 Configuration Options	Lists various customization options that may be used to personalize APCBASIC to non-standard situations.
7.4 CP/M System Differences	Describes all the differences in using the CP/M APCBASIC over the North Star DOS implementation of APCBASIC.
7.5 Version Change Histories	Describes all the changes made to APCBASIC through the various upgraded versions. Mostly useful to those users moving up from an earlier version to a later one.
7.6 Implementation Notes	Details on the internal operation of various APCBASIC activities.

-- 7.1 ERROR MESSAGES --

This section describes all error types and messages handled by APCBASIC. Most of the detected errors may be trapped by the APCBASIC program with the ERRSET statement described in Section 5.4 and have a type code associated to each. The remaining error messages have no type and are always issued when their corresponding error occurs. It is not possible or feasible to recover from errors of this type and ERRSET traps are ignored if they occur. Fortunately they are usually revealed during the debugging phase of program development and do not occur in well tested final versions of programs.

NON-TRAPPABLE ERRORS:

M E S S A G E	D E S C R I P T I O N
Continue Error	Attempted to CONTinue execution of a program without being in the state of temporary suspension left after a CTRL-C or programmed STOP.
Loading Error	Attempted to LOAD a APCBASIC program file that either doesn't contain a valid program or the program is too large to fit the memory available.
No Program Error	Attempted RUN without a program or a $ reference was used in a command when no program existed.
Double Def Error	Two or more function DEFinitions exist for the same function name. This can only occur at the beginning of program execution prior to executing the 1st statement of the program.
Memory Full Error	All memory space allocated to APCBASIC has been consumed. This can be caused by DIMensioning strings and arrays too big, using recursive GOSUBs or FNs that don't terminate, or performing operations that temporarily use more memory than is available. String expressions require enough temporary memory to hold the entire result, for example.
Return Error	RETURN statement encountered without an active GOSUB or FN underway.
Exit Error	Looping structure (FOR or WHILE) not present when Exit statement encountered.
Missing Next Error	No closing NEXT statement was found for a WHILE or FOR statement being initialized.
Next Error	Encountered a NEXT statement without an opening FOR or WHILE statement.
FN Def Error	Either a reference to an undefined function was encountered, or a function was DEFined within another function DEFinition.

Exprn-Depth Error	A result of too many nested levels of parentheses in string or numeric expressions. About 15 levels of parentheses are available.
Local Error	LOCAL statement encountered inside a loop (FOR or WHILE) or outside any calls to GOSUBs or FNs.
User Trap Error	ERRSET #<error type> statement encountered without an error trap in effect (as set by ERRSET <line>).
Translator Error	APCBASIC converts a program to a faster internal form prior to RUNning it. This error is generated when this task could not be completed for some reason, most likely caused by insufficient memory.
Buffer Update Error	Disk error encountered when attempting to update file buffers at program termination. No line number is associated with this error. The offending file is CLOSEd without updating its buffer, losing all new data it contains (512 bytes maximum). Other error messages may immediately precede this one.

TRAPPABLE ERROR TYPES:

TYPE	MESSAGE	D E S C R I P T I O N
1	Argument Error	An invalid argument was supplied to a command. Arguments of the wrong data type will generate error #4: TYPE ERROR.
2	Re-Dimension Error	Attempted to DIMension an existing string or array within a User-DEFined function (legal elsewhere).
3	Out of Bounds Error	A numeric value was outside the permitted range or file operations attempted to extend past the end of the file or disk. This is often due to array or string subscripts out of range.
4	Type Error	Data specified for an operation was of the wrong type: usually a string (or number) was given where a number (or string) was expected. Also indicated if an endmark is encountered during a READ#.
5	Format Error	Unknown or impossible numeric format was specified, such as a width specification not greater than the decimal specification.
6	Line# Error	A line number refers to a non-existent line or an ASCII line was ENTERed from a file without a line number.
7	File Error	An improper file operation, file number or file name was specified.

8	Disk Error	Unreadable or non-existent disk area was accessed. Also from writing to a write-protected disk.
9	Div/0 Error	An attempt was made to divide a number by zero.
10	Syntax Error	Improperly constructed APCBASIC statement, command, or expression.
11	Read Error	Attempted to access a DATA list when none existed.
12	Value Error	A string supplied to the VAL() function was not a proper numeric representation.
13	Mismatch Error	User-DEFined function specified with the wrong number of arguments.
14	Numeric Ovfl Error	Arithmetic operation attempted to produce a value of 10^62 or larger.
15	Stop	The running program has stopped in a CONTinuable state. This is caused by STOP statement or a CTRL-C from the console to interrupt program execution and trappable (as "error" type 15) for various applications with ERRSET.
16	Length Error	An INPUT line was typed which exceeded 159 characters or a CHANGE command lengthened a program line past 159 characters.

— 7.2 APCBASIC ALTERNATE KEYWORDS —

APCBASIC represents its reserved keywords internally as single-byte codes. When LISTed, these are displayed as standard keywords which however may not be the original keyword entered. For convenience and in some cases for brevity, APCBASIC will accept several different keywords for the same thing while it displays only its primary form. For example all of the keywords found in North Star BASIC are accepted by APCBASIC on input but display in their corresponding APCBASIC keywords. All such correspondences are listed below:

PRIMARY	ALTERNATE KEYWORDS
RENAME	REN
DOS	BYE
SAVE	NSAVE
DELETE	DEL, CLEAR, SCR
DIR	CAT
EDIT	ED
CHANGE	REPLACE, CH
TRACE	TR
LIST	LI
ENTER	AUTO, ENT
MERGE	APP
BY	STEP
PRINT	! (Both LISTable)
GOTO	GO TO (Significant Space)
NOMARK	NOENDMARK
RETURN	RET
LINK	CHAIN
LOG	LOG10
LN	LOGE
INCHR$	INCHAR$
POS	PTR
>=	=>
<=	=>

-- 7.3 CONFIGURATION OPTIONS --

Your version of APCBASIC can be personalized in several respects to accomodate its operating environment. These options are implemented by making changes to certain memory locations within APCBASIC. The CONFIG utility program provided in the APCBASIC software package permits interactive option selection and updates your various copies of APCBASIC to reflect your desired changes. All the options available for both North Star and CP/M versions are given below (check for applicabiltiy to your system). Details on location and format are not provided since the CONFIG program (usage instructions in Section 8.3) eliminates the need to know these. You can re-configure APCBASIC whenever necessary.

FLOATING POINT BOARD ADDRESS

The standard location of EF00 Hex may not be suitable in your situation. Any address of the form XY00, where X is any Hex digit and Y is 3, 7, B, or F is permitted. Check your system documentation to determine the proper value. Applies only to versions supporting the North Star Floating Point Board.

CONSOLE BACKSPACE SEQUENCE

When you strike a backspace (CTRL-H) or DEL key on the console keyboard, APCBASIC sends a standard backspace-space-backspace (ASCII 8,32,8) to the console screen to backup the cursor and erase the previous character. On rare CRTs or on hard copy consoles this may be unsuitable. You may specify any sequence of 1 to 4 character codes for performing this function. Consult your terminal manual to determine the proper sequence.

HIGH MEMORY ADDRESS

Normally, APCBASIC uses the highest available memory address above its load address to always allocate the maximum amount of memory for its use. In some situations, portions of high-memory are used for special purposes during APCBASIC execution, precluding its use by APCBASIC. You can establish any memory address (specified in Hex) to be permanently set as the high-memory address, however it should be as high as possible to permit the maximum program and data size. See your system documentation for the proper address where applicable.

SYSTEM INTERRUPTS FLAG

In CP/M systems, you must inform APCBASIC versions that use the North Star FPB whether or not interrupts are used during APCBASIC operation (required for proper arithmetic operation). North Star systems already have a special system flag set aside for this purpose (RWCHK at DOS+2BH) and no corresponding personalization is necessary.

INITIAL STATE OF CTRL-C DETECTION

Normally APCBASIC will always detect a CTRL-C typed from the console and abort program execution. Using PARAM(1) you can disable or re-enable this mechanism. However for special 'turnkey' systems of APCBASIC programs, CTRL-C should be disabled at startup (prior to executing the 1st program statement). The state of the CTRL-C detect at startup can therefore be configured as a separate option. Of course PARAM(1) may be later executed within the program to modify this state as needed.

CUSTOM DEVICE I/O DRIVER SOFTWARE

The eight I/O devices supported under APCBASIC are generally the concern of the operating system. Under North Star systems, you can personalize the DOS to modify the I/O drivers used by APCBASIC. However under CP/M, only limited I/O flexibility exists and does not normally support 8 I/O devices. Because of this, APCBASIC devices 1 through 7 (0 cannot be changed) may be re-assigned to use your own I/O driver software already present somewhere in known memory locations (CP/M versions of APCBASIC only).

Each device number can be assigned two jump addresses: one to an input subroutine and one to an output subroutine. These routines may freely use any of the registers for any purpose and return to APCBASIC via a RET instruction. Input subroutines must return the character in the accumulator (ACC). Output subroutines expect the output character in the E register when entered. No other restrictions apply and reasonable use of the stack (SP) is permitted.

-- 7.4 APCBASIC VARIATIONS UNDER CP/M SYSTEMS --

Certain changes are necessary to conform to the conventions and facilities provided under the CP/M operating system. These are described below with the assumption that you understand general CP/M usage and will refer to the main instructions for further details of the features below. Only the differences between APCBASICs under CP/M and North Star DOS are described.

File Names (for Programs & Data)

APCBASIC programs must be in files of type '.ZBA'. Commands specifying program files names do not however include the file type because '.ZBA' is always assumed and inserted automatically (commands: LOAD, SAVE, MERGE, LINK). All other file operations require unambiguous generic file names which bear the file type with the name in all file references (operations: CREATE, DESTROY, OPEN, FILE(), RENAME). Remember that ambiguous file names (those containing '?' and '*') and names improperly formed (eg. too long) cause an immediate FILE ERROR when used. General file names that do not specify (eg. "XYZ" instead of "XYZ.TYP") will have a default type consisting of three spaces (blanks).

Drive References in File Names

Absent drive reference always implies the default. Initially 1, this may be changed with the PARAM() statement. Drive numbers range from 1 to 15 in functions, statements and commands, and 'A' to '0' in file names. Zero if specified implies the current default drive#. Characters '0' thru '9' may specify file name drive references instead of 'A' through 'J', as well as lower case letters 'a' through 'o' (only the least significant four bits of the drive designator are used).

In file names, the drive reference may be placed in front of the name or appended to the end of the file name. For example the file names "B:TFILE.DAT" and "TFILE.DAT,2" and "tfile.dat,b" all refer to the same file on drive 2. This flexibility permits use of North Star file names in programs running under CP/M APCBASIC.

Due to a serious design ommission in the CP/M operating system, APCBASIC cannot regain control after a BDOS SELECT ERROR. It is therefore imperative that your program NEVER attempts to access a file on a non-existent or otherwise unavailable drive. Other BDOS errors, such as a hard disk error, are usually infrequent but will also send you back to the CP/M operating system level without giving your program a chance to recover (thanks to DIGITAL RESEARCH). This is a universal problem with all programs that run under CP/M.

OPEN, CREATE

OPEN #<file number>,<file name> opens an already existing file for general file operations under the file number given (ranging 0..15). No numeric file type and no file size variable is specified as in North Star APCBASIC.

CREATE <filename> creates a new file of initial size 0. Note that file types and drive references are included as part of the file name where applicable. A trappable (with ERRSET) error occurs with attempts to create files already in existence.

READ, WRITE

End-of-File processing must be done explicitly when the CP/M end-of-file mark is used in the file being processed. Attempting to READ past the last file block (128 byte units) written results in an OUT OF BOUNDS ERROR. Writing past this point will extend the file size and does so in increments of 512 bytes at a time. All files produced by CP/M APCBASIC will be a multiple of 512 bytes, but can READ files with any multiple of 128-byte file blocks, but always extends a file by 512 bytes at a time with the WRITE statement.

FILE(), FILESIZE(), SPACE()

FILE(F$) returns 0 if the file name string expression F$ is not found, and 1 if it does exist. This is different from the North Star FILE() function which returns a numeric file type (or -1 if not found) instead.

FILESIZE(<filenumber>) returns the number of blocks (256 bytes) of virtual file space used in the file number specified. This differs from the unchanging file size implemented by North Star DOS. Note that the block size is identical with all versions of APCBASIC.

SPACE(<drive ref>) returns the number of file blocks still available on the drive specified by the numeric drive reference. Drives 1 to 15 are permitted with 0 referring to the current default drive. The implementation supports block counts up to the floating point integer precision size.

DIR

A third parameter to DIR (both command and statement) may optionally be given to specify a file type selection. Only those files of that type are displayed in the directory listing. The file types are not shown on such listings to provide a neater, more compact display. This type parameter must be specified as a string expression, for example: DIR 2,"ZBA" (lists APCBASIC programs on drive 2).

CALL(5)

The CP/M disk system must be reset after diskette changes by using the CALL(5) function. The sequence of operations is as follows: CLOSE all files; direct the user to make all diskette changes and press return when done; reset the disk system [eg. R=CALL(5) would do it] and (RE)OPEN all files required from that point.

Device I/O

The eight input devices are assigned as follows: 0,1,4-7=Direct Console Input, 2=Reader, 3=CP/M Console Input.

The eight output devices are assigned as follows: 0,4-7=Direct Console Output, 1=List Device, 2=Punch, 3=CP/M Console Output.

You can 'patch in' jump addresses to your own I/O driver routines for any or all Input (0-7) or Output (0-7) devices. This process is done with the CONFIG program described in Section 8.3. The I/O drivers themselves must already be somewhere in memory at known locations (like in the BIOS user area).

MOVING NORTH STAR BASIC FILES TO CP/M

If you are using CP/M APCBASIC on a North Star system and you have North Star type 2 files containing programs in BASIC, the following procedure may be used to move those file to your CP/M environment:

(1) Determine the number of blocks in your program by LOADing it into BASIC and using the SIZE command. Do not use the file size given in the DOS directory because it is larger than the program size.

(2) From the DOS load your file into memory at 3000H: LF filename 3000

(3) Boot in your CP/M system and then get into DDT.

(4) Move your program memory image down to 100H with the DDT command: M3000,8000,100 where 8000 represents a memory address at or above the end of the memory image.

(5) Type CTRL-C to get back into the CP/M command level.

(6) Save the memory image at 100 with the command: SAVE N FILE.ZBA where N is the number of blocks determined from step (1) and FILE.ZBA is any file name with type .ZBA necessary for all APCBASIC programs under CP/M.

— 7.5 VERSION CHANGE HISTORIES —

Periodically, extensions are added to APCBASIC that create a more powerful or otherwise improved new version. New releases of APCBASIC will continue to support programs written under earlier ones. Programs that take advantage of the latest extended features will not execute properly under earlier releases of APCBASIC.

Although this manual is complete through Release 2.4, if you were upgrading from 2.1 to 2.4 it would be difficult to determine the differences without comparing the entire manual with an earlier one. Therefore this section gives brief descriptions of all the modifications included with each new release of APCBASIC.

CHANGES LEADING TO VERSION 1.09:

(1) Faster RUN version. Compiles program to fast binary internal form just prior to execution. Totally transparent to the user.

(2) Minor syntax changes: multiple character variable, array & function names must be contiguous (no embedded spaces); the optional LET token is no longer supported.

(3) Function DEFinitions must appear as the 1st statement on the line in which they are defined.

(4) EDIT search strings may contain question marks (?) as 'wild card' characters that match any single character.

(5) LIST & DEL may use a dollar sign ($) to denote the last line of a range of lines. (Eg. LIST 1,$ or DEL 123,$)

(6) A third string argument may appear in the LIST command. Similar to EDIT, the string causes selective display of only those lines matching it.

(7) The TRACE IF <exprn> command was added for conditional debugging. It provides the usual TRACE functions whenever the <exprn> evaluates to a non-zero value. When zero, your program executes normally until the condition again becomes true.

(8) NSAVE has been absorbed by the SAVE command. When SAVing a program, APCBASIC will respond with an OLD FILE or NEW FILE message requiring user verification (Y or N response). It proceeds only upon a 'Y' response. In addition, APCBASIC remembers the previous file name SAVEd and supplies this file name for subsequent SAVEs if you do not give a file name in the SAVE command (ie. SAVE followed by a carriage return).

(9) Overlay structures are supported using the dynamic MERGE and DELETE statements. MERGE will overlay selected portions of the running program with new modules. DELETE will delete selected portions of the running program that are no longer needed. Any existing line range that includes the DELETE statement itself is permitted. All variables and OPEN files are preserved under both statements.

(10) The system variable INDEX always contains the list position selected by a previous MIN(..) or MAX(..) function. Use it as any other variable.

(11) System variable PI returns its namesake rounded to prevailing precision.

(12) The MOD(N,M) function returns the smallest positive value V such that N-V is divisible by M. This is different than the 'straight' remainder when negative args are supplied.

(13) CEIL(X) returns the lowest integer greater than or equal to X.

(14) TRUNC(X) returns the 1st integer encountered when moving from X to zero.

(15) SGN(X,Y) returns ABS(Y) with the sign of X. SGN(X) remains unchanged.

(16) ROUND(X) returns X rounded to the nearest integer. ROUND(X,P) rounds X to a maximum precision of P digits, where INT(P)>0.

(17) TAN(X) returns the tangent of X given as an angle in radians.

(18) SPACE(D) returns the number of blocks (256 byte blocks) remaining on disk drive D. If D=0 then the default drive is assumed. The feature in the OPEN statement providing the same information is no longer supported, ie. the file size variable will not be set to the #blocks remaining if the file is not found.

(19) After a CTRL-C STOP in the RUN version, control passes back to the DOS. By jumping to GO ADDRESS + 14H you may CONTinue from where you left off.

(20) The PARAM(P) statement allows control of several internal execution factors. It may be used on the left side of an assignment statement (=) to assign new values, or on the right side to determine their current setting. Expression P currently selects parameters 0, 1, ..., 4: 0=version number, 1=CTRL-C disable, 2=default drive#, 3=default I/O device, and 4=prevailing numeric precision. All may be 'read' but only 1, 2 and 3 may be set. Expect further parameter definitions in later releases.

(21) Programs can RENAME files using the statement: RENAME <oldname>,<newname>. Both file names are given as string expressions. APCBASIC reports a FILE ERROR if the <oldname> doesn't exist or the <newname> already exists.

(22) Global editing is further enhanced with the CHANGE command that replaces one string with another everywhere or selectively within a line range. Use the command form: CHANGE <line1>,<lineN>,<oldstring>,<newstring> The line range & the <oldstring> are identical with the extended LIST command. All four arguments are mandatory. APCBASIC will request 'VERIFY?' to allow user control of each replacement ('Y' response). An 'N' response causes replacement of all occurances found.

(23) REV$(<string exprn>) returns a reversal of the string argument supplied.

(24) TRIM$(<string exprn>) returns the given argument stripped of leading & trailing spaces.

(25) String expressions now include bit-vector logical combinations using the operators: NOT, AND, OR, XOR, and EQV. Each binary operator logically combines the corresponding bits of two strings into a new result string. The NOT unary operator reverses each bit of the string that follows it. If the argument strings differ in length, the shorter of the two will determine the extent of the operation and hence the length of the result. The string operator precedence follows in highest-to-lowest order:

(0) String vbls, Quoted text, String functions
(1) Strings in parentheses
(2) NOT
(3) String Factors (*)
(4) Concatenation (+)
(5) AND, OR, XOR, EQV (Descending Precedence)

Any ordering of operations may be effected using appropriately placed parentheses. Note also that string factoring must be enclosed in parentheses when it is not the last term of the string expression to avoid ambiguity (between numeric & string operators).

Be most careful using complex string expressions in string comparison operations. It is the users' responsibility to ensure that mixed string and numeric expressions are sufficiently parenthesized to resolve any inherent ambiguities.

(26) XOR and EQV may be used for numeric logical expressions. They act on the logical values (zero, nonzero) of their arguments. Thus A XOR B is true only if one arg is zero and the other is nonzero; A EQV B is true only if both args are zero or both ars are nonzero. Logical operator precedence ordering is as follows: AND, OR, XOR, EQV.

(27) Polynomials from degree 1 to 255 may be evaluated using the POLY(X,A(),D) function described in Section 6.2.

(28) Strings of any length may be rotated left or right by 0 to 255 bit positions at once using the ROTAT$(S$,R) string function described in Section 6.3.

(29) Re-direction of Device I/O to and from files is made possible using 'devices' 8 thru 15, which are assumed to be file numbers previously assigned with the OPEN statement. This allows commands and statements that use #device numbers (notably PRINT, INPUT, and LIST) to direct their data transfer operations to files instead of the usual devices. See Section 5.2 for details.

(30) The DEL command requests user verification before erasing the entire program (when no line range is given).

(31) The CHR$() function may have two arguments to generate a sequence string of ascending ASCII codes. See Section 6.3 for details.

CHANGES LEADING TO VERSION 2.1

(1) Numerous additions to the formatting capabilities permit much greater flexibility. In particular, the width (w) specification (wI, wEr, and wFr) may be omitted to suppress all but one leading space (as in free formatted values). The wI (or I) format rounds all non-integers before printing the number and is no longer considered a FORMAT ERROR. Finally a T format modifier may be employed for trailing minus signs (or plus signs), which are always placed as the last character of the field, regardless of any zero-suppression in effect. See Section 5.2 for further details.

(2) The string function TRAN$(A$,S$,R$) returns A$ after translating any of its characters matching those in S$ to their corresponding characters in R$. All three arguments may be general string expressions.

(3) APCBASIC will raise positive or negative numbers to integer powers from -9999 to 9999. An OUT OF BOUNDS ERROR occurs when negative numbers are raised to non-integer powers or powers outside the range above. APCBASIC computes integer powers much faster than non-integer powers.

(4) The CRUNCH utility reduces programs about 10% further than previously. Both CRUNCH and ZBIG have an easier calling sequence; see Section 8.0 for further details.

(5) Direct statement execution can include any statement and may CONTinue via a direct GOTO, EXIT <line#>, RETURN or RETURN <result>. See the discussion of direct statements in Section 2.3.

CHANGES LEADING TO VERSION 2.2

(1) The FOR statement can now be defined as: FOR <vbl>=<range1>,<range2>,... where each range can be one of three forms:

(1) <exprn1> TO <exprn2>
(2) <exprn1> TO <exprn2> BY <exprn3>
(3) <exprn1>

Form (3) specifies the special case range consisting of one value. When the current range runs out, APCBASIC accesses the next range in the list, evaluates all its expressions, then continues loop execution under its new parameters.

(2) The FIND function was added specifically for searching large string tables. Use the form:

FIND(<string vbl><comparison operator><string exprn>,<increment>)

For example: FIND(T$>" ",4) returns the position in T$ of the first character greater than a space (ASCII 32), when comparing every 4th character. The value returned is always relative to the beginning of the string variable, even if it is indexed for a smaller, localized search. Any of the comparison operators may be employed (=, <>, >, <, >=, or <=). When the relation cannot be satisfied, zero is returned. The search string expression must evaluate to a string not exceeding 255 characters long. The increment is optional and defaults to 1 when omitted. Thus as a special case, FIND(T$=A$) is equivalent to MATCH(T$,A$).

(3) Errors in a RETURN <exprn> statement are now recoverable using the appropriate ERRSET statement. This used to prevent the FN from RETURNing normally after any attempts to recover from a RETURN error. An obscure problem resulting from sequential parameter binding in FNs has also been fixed. All FN arguments are now fully evaluated before binding them to their corresponding parameter variables.

(4) Error detection has also improved to more accurately reflect the type of error and its origin location in APCBASIC programs. In particular, many TYPE ERRORs and MISMATCH ERRORs were erroneously reported as SYNTAX ERRORs, and certain errors involving FNs would report the wrong error line number.

(5) To further assist debugging efforts, the TRACE RET command was added. This command will display the RETURN path (in line numbers) after the program stops for any reason (eg. CTRL-C, STOP, program error, etc). If the program is CONTinuable, the first line number displayed will be the point of CONTinuation. The RETURN path is displayed all the way back to the first FN or GOSUB reference that began the sequence. Since this may be quite long, you can abort this display by typing a CTRL-C. (GOSUBs and FNs can descend hundreds of levels in APCBASIC)

CHANGES LEADING TO VERSION 2.3:

(1) Multi-DIMensional string arrays may now be defined and accessed. Specify string array DIMensions just like numeric arrays except that you must supply the maximum length of each array element for the last value of the DIMension list. For example, DIM B$(7,20,16) defines an 8 by 21 (zero based array subscripts), two-dimensional string array whose element strings may range from 0 to 16 characters each. You must always refer to B$ with a subscript list to indicate a specific array element, as in: R$=B$(I,J); R$=B$ would generate a SYNTAX ERROR. Specify only the array DIMension positions; leave off the length parameter. You may index string array elements by appending the index specification, as in: B$(I,J)(K), B$(I,J)(K,L) or B$(I,J)(K:3). Unlike numeric arrays, string array variable names cannot be assigned to scalar strings too: they must be unique.

(2) To re-initialize any variable (string or numeric), list them in the following statement: RESTORE X,Y,A(),B$,R$(). The empty parentheses indicate array variables. All numeric variables are set to zero and string variables are filled with spaces. This usage has no relation to the standard RESTORE statement used with DATA statements.

(3) You can read console (keyboard) characters without waiting, by specifying any negative value for the device number D in INCHR$(D). INCHR$(-1) always returns a one-character string, even if no console character was typed. So you must test the result character for the no-character-typed code. CP/M versions return an ASCII 0, but other operating system system conventions are different and this code, although consistent, must be determined specifically in your operating environment (eg. under North Star DOS). You must disable the CTRL-C detection system to use this feature (PARAM(1)=1 would do it) because it immediately swallows any character appearing from the console.

(4) SWAPDEF A(),B(),C$,D$,... will swap the variable names instead of their contents as in the SWAP statement. This may be employed to pass arrays and strings to GOSUBs (or FNs) by easily substituting them for the variables used within the GOSUB procedure, then SWAPDEFing them back again after the RETURN. The variables are indicated under the same conventions for specifying a LINK common variable list.

(5) SWAP statements can now include a list of variable-pairs to SWAP. Separate each pair with a comma (,). For example: SWAP X,Y,A$,B$,R4(J),Z(I,K). A SYNTAX ERROR results from an odd length list. A TYPE ERROR results from attempting to SWAP strings with numbers.

(6) The OUT statement will accept either numeric or string data for output through CPU ports. For example: OUT P,C$ will output the 1st character in string variable C$. Any general string (or numeric) expression may be specified, however only the 1st character of the string is OUTput. If a null string is specified, an undefined value is OUTput.

The INP(X) function reads a byte value from port X and returns either a numeric result or a character string result, depending on the context in which it is used. Used where a string is expected, INP(X) returns a 1 byte string. Otherwise, a numeric (floating point) result is returned.

These expansions to INP() and OUT operate faster, due to fewer conversions between character and floating point format, and permit access to the bit-masking operations provided by the logical string operators: AND, OR, NOT, XOR, and EQV.

(7) Two new read-only PARAM() codes have been defined. PARAM(5) returns a code specifying the operating system environment under which APCBASIC is executing (0 for North Star DOS, 1 for CP/M, 2 for APC's MTOS). PARAM(6) returns the address to the original command line typed that loaded APCBASIC and a program for immediate execution. Zero is returned if no auto-run program was specified. Use this feature to pass additional parameters to your program from the calling command sequence with the statement: EXAM PARAM(6),L$. Your program must perform all necessary decoding; the line always ends with a carriage return code (ASCII 13). Since some systems (eg. CP/M) reuse the input area for other things, be sure that your program reads this parameter line at the start of program execution.

(8) A CALL statement (ie. not the function) has been added to permit full access to the 8080 register set. Use the form:

CALL <address exprn>,<data register exprn$>[,<result register vbl$>]

Register values are specified as characters in the string arguments and are positionally defined: ACC, F, B, C, D, E, H, L. The data registers may be a string expression of any length. The result register string vbl must be at least 8 bytes long to hold all returned register values (when this optional parameter is supplied). Use BIT(), ASC(), and CHR$() functions to pack & unpack your desired values to & from the string arguments.

(9) CP/M versions of APCBASIC will accept North Star file name formats to promote compatability between both systems. In particular you may always specify a drive number at the end of a file name. For example, the file names B:TEST and TEST,2 are equivalent. APCBASIC uses the lower four bits of the drive character to determine the drive, permitting letter or digit designations (eg. TEST,2 is the same as TEST,B).

(10) The DIR statement (and command) supports a third parameter to specify a file type selection (under CP/M versions only). When supplied, only the files that match that type are displayed. This parameter is specified by a string expression that evaluates to the desired 1-3 character CP/M file type. For example: DIR 2,"COM" will display all .COM files on drive #2. The file type expression must follow an explicit drive number designation. For example: DIR "ZBA" and DIR #1,"ZBA" are illegal.

(11) CP/M versions can now accept files with any number of CP/M file blocks. Attempts in prior versions to read the last few (128 byte) file blocks of a file whose current size was not a multiple of 4 blocks, would result in an OUT OF BOUNDS ERROR. APCBASIC always increases file sizes by 4 (128 byte) blocks at a time.

(12) CP/M versions now properly detect an out-of-disk-space condition and end with a DISK ERROR. Prior versions produced unpredictable results.

(13) The cross-reference utility (ZBIG) will selectively display only those entries which have references to them in a user specified line range. Type the line range in the ZBIG calling sequence immediately after the source file name: ZBIG <program> <1st line> <last line> Omitting the line range defaults to the entire program; omitting the <last line> will reference a line range consisting of the 1st line only.

(14) The program size reduction utility (CRUNCH) permits user control over the line-joining process it performs. Just answer Y or N (Yes or No) to the CRUNCH request: JOIN LINES WHERE POSSIBLE? Remember that after lines are joined, you may not be able to EDIT the program in APCBASIC.

(15) After a LOADING ERROR or MEMORY FULL ERROR when LOADing a program, APCBASIC will maintain as much of the program as it could, rather than clearing memory.

(16) To permit commas (,) and lower case characters in string arguments of the CHANGE command, any string argument may be surrounded by quotes (""). For example: CHANGE 1,$,"A(I,J)",Z(R) will change A(I,J) to Z(R). Such quotes may not be used as ordinary characters in either string.

(17) The software floating point multiply and divide (ie. in APCBASIC versions not requiring a hardware processor), has been significantly optimized. They now run 2-4 times faster than the multiply and divide found in standard North Star BASIC, depending on which precision you are using.

(18) The TRAN$() function now runs many times faster than the originally introduced version, particularly on large strings. However, the new method requires additional workspace for this improvement, and erases the prior content of the editing 'old line' buffer. Neither replacement string (2nd and 3rd arguments) may exceed 256 characters without causing an OUT OF BOUNDS ERROR.

Changes leading to Version 2.4:

(1) String variable indexing methods are no longer limited to just string variables. Any term of a string expression may now be followed with a indexing expression, as in: STR$(X)(2), "ABDCEFG"(2,6), or FNA$(I)(K:L). Apply the same rules that you use for string variable indexing. To index a complex string expression, surround it with parentheses then append your indexing expression, as in: (A$+B$+C$)(J,K). Index expressions may be appended one after another to further subindex a string, for example:

"ABCDEFGHIJK"(3,8)(4:2) = "CDEFGH"(4:2) = "FG"

An additional indexing mode has been added to the string indexing capabilities to directly index a right-substring. Suppose you wish to access the last N bytes of A$. Previously, you only had two options: A$(LEN(A$)-N+1,LEN(A$)) or A$(LEN(A$)-N+1:N). The new method is much simpler and faster: A$(:N), where the colon (:) as the first character signifies this mode and N is a length. This notation may be used for either string variable or expression indexing.

To streamline single character indexing, the length value following the colon (:) defaults to one (1) when omitted. Thus A$(K:1) may now be expressed as A$(K:), saving execution time on evaluation of the length parameter. An additional benefit of this is that the last character of string A$ may be indexed as A$(:), which follows from the preceding paragraph.

(2) The context string search used in LIST, EDIT, and CHANGE commands has been generalized to match letters without regard to their upper or lower case status (eg. NEXT=next=NeXt etc). APCBASIC keywords embedded within string arguments to these commands are no longer translated to internal form, which in the past altered the intended effect of certain strings.

The CHANGE command no longer permits access to the line number portion of program lines, due to its dangerous implications. Be sure to use the VERIFY option after you specify a number search in a CHANGE command, as it may match portions of line numbers in GOTOs, GOSUBs, etc. Changing line numbers should only be attempted with the REN command.

(3) All APCBASIC messages have been re-written in upper/lower case format to provide a more human, rather than DATA PROCESSING, appearance.

(4) IF statements may employ a compound statement for the THEN or ELSE clause. A compound statement is one or more statements, separated by semi-colons (;), and surrounded by brackets []. The entire IF statement is limited to what can be fit into a program line (159 characters maximum). FOR-NEXT loops can therefore be used as a THEN or ELSE clause, as long as the entire loop fits within the brackets.

(5) A WHILE statement has been added to enhance the looping capability of APCBASIC. It works just like a FOR statement, except that the FOR initialization statement (eg. FOR I=1 to 100) is replaced with:

WHILE <condition>

where <condition> can be any numeric expression, comparision, etc. This condition is tested at the 'top' of each loop and the loop terminates when it becomes false (zero). Remember that unless the body of the loop causes the condition to go false, WHILE loops will never terminate. You can also terminate WHILE loops with an EXIT statement in exactly the same manner as in EXITing FOR loops.

The WHILE token in APCBASIC is the same one used in North Star BASIC for the optional LET token (eg. LET X=0). Thus WHILEs found in existing North Star BASIC programs LISTed in APCBASIC must be deleted.

(6) The TRACE mode now waits until a valid TRACE control character is typed and 'rings the bell' to indicate invalid controls. Several new controls have been defined as follows:

ESC	Terminates the TRACE mode and continues program execution. This is the only way to terminate the TRACE mode.
SPACE BAR	Executes the next statement and breaks (single-step mode). This was formerly done by 'any character'.
^	Resumes program execution until the current sub-program or loop terminates (GOSUB, FN, FOR-NEXT, WHILE-NEXT).
CTRL-T	Resumes program execution until a line number transfer occurs, such as a GOTO, GOSUB, ERRSET trap, etc.

To set the single-step mode immediately, enter the TRACE command without any arguments (which used to terminate the TRACE mode). For debugging convenience, if a LINK to program occurs while the TRACE mode is active, no automatic size reduction (space & REM removal) will take place. Once invoked, the TRACE mode persists until terminated with the ESC control. LIST, EDIT and TRACE RET no longer deactivate the TRACE mode.

(7) ERRSET #<type> may be used to generate an error of the given type, for purposes of debugging and special program control applications. You must have an active ERRSET in effect at the time this statement is executed, otherwise a USER TRAP ERROR is issued. For example: ERRSET #7 generates a FILE ERROR exactly the same way that OPENing a non-existent file would. The <type> can be a numeric expression that evaluates to a value between 0 and 127.

The ERRSET variables (eg. X and Y in: ERRSET 4000,X,Y) are now optional. These must be omitted from right to left when not specified, ie. if the 1st variable (for the line# of the error) is omitted, then you cannot specify the 2nd variable (for the error type code) either.

(8) LOCAL statements may now be used directly in User-DEFined FNs, the way they were previously available for GOSUBs only.

(9) The ASC() function now accepts general string expressions, functions, or literals (string constants), in addition to just string variables. It still returns the ASCII code of just the first character, so long expressions are unnecessarily redundant and will execute more slowly.

(10) The INTERNAL STACK ERROR now displays as EXPRESSION DEPTH ERROR. This is what that error really meant all along. The CONTROL STACK ERROR message has been replaced by one of four more meaningful error messages depending on the context:

NEXT ERROR	NEXT Statement without a preceding FOR or WHILE
LOCAL ERROR	LOCAL Statement inside loop or outside GOSUB or FN
EXIT ERROR	EXIT encountered outside all FOR of WHILE loops
RETURN ERROR	RETURN Statement encountered outside all GOSUBs and FNs

Re-Dimensioning string or numeric variables is now illegal within User-Defined Functions (FNs), directly or indirectly. Such an occurance generates a RE-DIMENSION ERROR (Type 2). You can however DIMension a new string or array within User-Defined FNs.

(11) You can enter a APCBASIC program from an ordinary symbolic text file with an extension to the ENTER command. First, you must OPEN the text file under a file number ranging from 8 to 15. Then, simply say ENTER #F, where 'F' is the file number assigned. At this point the text file is input from the file and each line is displayed on the console. This process continues to the end of the file according the the following rules:

(a) Each line is terminated with a carriage return (ASCII 13).

(b) When a line-feed and a carriage return appear in pairs, the 2nd character of the pair is ignored.

(c) Empty lines, consisting of only a carriage return, are ignored.

(d) A control character (ASCII codes 0 - 31) or any character with bit 7 on (ASCII codes 128 - 255) as the 1st character of a line signals that the end-of-file has been reached.

(e) Control characters elsewhere in the line will appear as question marks (?) and will have to be removed in a later editing phase.

(f) Each and every line must begin with a line number in the range 0 to 65535. Otherwise a LINE# ERROR is issued and the input process terminates. APCBASIC inserts each line into its proper sequence according to its line number, so they need not be in order.

(g) Lines are limited to a maximum of 159 characters. Additional characters beyond this limit are ignored and lost.

(h) The input process may be interrupted by typing a CTRL-C. To resume input after a CTRL-C or a LINE# ERROR, just retype the original ENTER #F command (possibly with a CTRL-G and a return).

(12) In all commands which accept line numbers (LIST, EDIT, CHANGE, DEL, TRACE, ENTER, but not REN), you may use a dollar sign ($) to denote the last line of the program. This notation, however, may not be used in any program statements. For example: DEL 10,$ or LIST $ or TRACE $.

(13) The version number returned from PARAM(0) takes the following form: U.VWX, where U=year digit, V=sequence number in that year, W=type of arithmetic processing used (0=software, 1=N* floating point board), X=0 for development version, X=1 for runtime version. Other indicators may be appended to this form from time to time.

After changing the default device number with PARAM(3), all command level console messages (READY, ERROR messages, etc) will continue to be sent to the console, rather than directed to the current default device.

PARAM(7) has been added to permit access to the ASCII code used to initialize strings and string arrays. At startup, PARAM(7) = 32, the ASCII code for spaces (blanks). You can revise this value to any code from 0 to 255 with an assignment statement: PARAM(7)=0.

PARAM(8) is another read-only parameter (cannot be changed) that returns the address of the 1st memory location above memory used by APCBASIC. Your APCBASIC can be made (using the CONFIG program) to leave some memory unused at the top for use by external machine code or data. PARAM(8) will always tell you where this is.

(14) INCHR$(X) will now input single characters from a file if the file is OPEN under file number X, where X ranges 8 to 15. This was added to be consistent with the other I/O redirection facilities in APCBASIC. Exactly one (1) character is read with each reference to INCHR$(X). No end-of-file mark is checked for, although an OUT OF BOUNDS ERROR will occur if you attempt to read past the physical end of the file.

Do not use INCHR$(X) in this manner directly in the data list of a WRITE statement to the same file, as it will upset the file pointer for the subsequent WRITE operation.

(15) Editing control characters which search for a typed character (ie. CTRL-D, CTRL-S, and CTRL-X) will, if that character is a letter, match either case (upper/lower) of the character. A new control, CTRL-B, will copy the rest of old line to the new line and then permit re-editing from the beginning of the line (equivalent to a CTRL-G-N sequence).

(16) When CTRL-C is typed in the command mode no STOP message will be issued, only READY. The STOP message is now given only to indicate interruption of a running program. The CTRL-C break character is recognized only from the console device and is just another control character when input from devices 1 to 7.

(17) The power operator (eg. X^Y) returns a value of 1.0 if both base and exponent are zero (ie. 0^0=1, whereas it used to evaluate to zero).

-- 7.6 NOTES ON APCBASIC IMPLEMENTATION --

To assist those implementing programs requiring detailed information on how programs are formed, saved and executed, this section should suffice for most applications. This information permits writing programs that might run faster, take less space or be less error-prone, by taking advantage of certain details.

PROGRAM INTERNAL FORM:

The first byte of the type-dependent information within the directory entry for type 2 files (APCBASIC program files under North Star systems), contains the number of file blocks of the file that actually contain APCBASIC program code. APCBASIC always adjusts this byte to reflect the actual program size when it SAVEs a program, and uses its value to determine program size when programs are LOADed. If for any reason this byte requires user adjustment, two successive TY commands (DOS command) can perform the task:

```
TY program 1 size
TY program 2
```

where 'program' is the program file name, and 'size' is the value you wish to overwrite the old value with (specified in Hex).

APCBASIC programs themselves consist of a sequence of specially coded lines and are terminated by a program endmark of 1 (byte value). After you type in each program line, APCBASIC reformats the line into the following format:

- The first byte contains the total number of bytes in the line.
- Bytes 2 & 3 contain the line number as a 16-bit integer.
- The remaining bytes contain the body of the line, which always ends with a carriage return code (0DH)

The various reserved words (eg. FOR, IF, GOSUB, GOTO, etc.) are all reduced to single-byte codes representing the reserved word, in the internal program line. References to line numbers are transformed into 16-bit integer format, and preceded by a flag byte (9AH) for identification. When programs are LISTed, APCBASIC converts these 'tokens' back into the standard reserved word, and line numbers back into ASCII integer representation. This is why programs created under other versions of North Star BASIC do not always LIST properly in APCBASIC. The few changes necessary for APCBASIC operation are easily done with the APCBASIC editing capabilities. See Section 9.1 for details on compatibility.

ARRAY ORGANIZATION:

String and numeric arrays are mapped into sequential memory locations in a specific pattern which should be understood when interfacing with machine code for array processing. The following example can be extended to any number of dimensions. Using an array DIMensioned: A(1,2,3), the following list of array subscript combinations is ordered by their sequential memory locations: 000, 001, 002, 003, 010, 011, 012, 013, 020, 021, 022, 023, 100, 101, 102, 103, 110, 111, 112, 113, 120, 121, 122, 123. Just remember that array subscripts vary from right to left, exactly like the digits of a automobile odometer.

DATA FILES:

Four fundemental data types are supported in APCBASIC files: North Star strings, North Star floating point format, 16-bit words and 8-bit bytes. The strings come in two types: short and long. Short strings have two leading bytes (a 3 value followed by a one byte length 0-255) followed by the string characters (one byte for each). Long strings have three leading bytes (a 2 value followed by a two byte length of 256+) followed by the string characters. Floating point numbers are stored on files in the same representation as operated on in memory (except for zero, which has its leading byte set to 10H). Their length is dependent on the precision of the BASIC they were written in, and must be read using a BASIC of the same precision.

16-bit integers are written low order byte first, high order byte second. 8-bit bytes are just written as is, in the order they appear. These binary formats cannot be detected by the TYP() function, which is designed only for strings, floating point values, and end-of-file marks. When floating point values are read or written in 8-bit or 16-bit formats, APCBASIC must convert between floating and binary formats, a rather time consumming process. When strings are read or written in binary byte format, no such conversion is required and operations procede literally hundreds of times faster.

APCBASIC performs file operations through a 512 byte buffer to increase throughput. When the same file is OPENed under several different file numbers, several buffers are assigned to that file. This permits certain random file operations to actually progress hundreds of times faster than could ordinarily be done with singly-buffered files. When APCBASIC reads data into a buffer from the file it first searches the other buffers to see if the desired data is already present in one of them. If it is, that buffer is assigned to the file and the operation continues without having to actually perform a file access to obtain the data. This eliminates redundant file operations and prevents buffer conflict in multi-buffered files.

THE SYMBOL TABLE:

Immediately after the program in memory is a table containing all the scalar variables, arrays, strings, user-defined functions, and file buffers. Called the symbol table, this structure is used for maintaining the working data and providing high-speed access to each data structure. Symbol access is dynamic in the sense that frequently accessed entries have faster access than seldomly accessed symbols. This is particularly true for programs containing enormous numbers of different variables (ie. 100 or more).

Because of the dynamic storage allocation of arrays and strings (re-DIMension capability), the symbol table will be reorganized to some degree after re-DIMensioning arrays or strings, dynamic MERGing, preserving data through LINK operations, FREEing file buffers and dynamic program line range DELETions. Therefore external support software using variable addresses (eg. assembly code matrix routines) must be used with care to ensure that data structures are located where those routines 'think' they are.

When FREEing file buffers or re-DIMensioning strings and arrays, those data structures and all those created after them will be physically moved in memory (nothing else is moved). After dynamic MERGing or DELETion, or communicating data between program LINKs, all data structures will be in memory locations different from before that operation.

THE CONTROL STACK:

At the upper end of memory is a highly transient data structure called the control stack, which is used for intermediate work space and various control structures. When numeric or string expressions are evaluated, the control stack maintains the temporary results that form the end result. In particular, when strings concatenated, multiplied or logically combined, the control stack must possess enough memory space to contain all the intermediate results until the final result is transferred to its programmed destination area. String expressions composed of large strings can easily consume all available memory before finishing. This causes an MEMORY FULL ERROR, which occurs when either the control stack or the symbol table expands into the other structure.

The control stack also maintains the return structures of GOSUBs and user-defined functions, as well as their local variables, DATA-READ pointer and local error control structures. This stack implementation of subroutines and local data permits unrestricted recursive programming structures to be fully realized. FOR..NEXT looping structures also make use of the control stack to maintain nested loops to virtually any depth.

**** 8.0 APCBASIC UTILITY PROGRAMS ****

This section concerns itself with several programs external to APCBASIC that perform functions useful to the development process.

Section 8.1	ZBIG Cross-Reference Index Generator. Shows the locations of each program structure used in your programs.
Section 8.2	CRUNCH Program Size Reducer. Reduces your program size by 30% and more through removal of immaterial spaces and REMarks.
Section 8.3	CONFIG: A configuration program to set various options in your versions of APCBASIC.

-- 8.1 CROSS-REFERENCE INDEX GENERATOR FOR APCBASIC --

In BASIC, it is not obvious what side effects are generated when lines are modified, variables re-assigned, or functions renamed. When programs get large, it is not clear whether a variable name is new or already in use, or how many times a given line number is referred to. Indeed, modifying BASIC programs written by someone else can be most frustrating without this information. If we only knew where to find all occurences of each program structure...

Written in Z80 machine code (8080 machines not supported), the APCBASIC Index Generator (ZBIG) provides the programmer with an instant directory to all user-defined functions, GOSUB's, variables, GOTO's, and other line referencing used in his BASIC program. ZBIG prints each label (name or line#) followed by a list of all lines that it appears in. These label entries are listed alphabetically within each of the following sections:

- o Numeric functions
- o String functions
- o Array variables
- o String variables
- o Scalar variables (non-subscripted)
- o GOSUB references
- o Misc. line references (GOTOs, etc)

ZBIG places no restriction on source program size or content except that it must be loadable in BASIC without error, and have reasonably correct syntax.

In recognition of the wide variety of hard & soft output devices available, a number of output controls are implemented to freeze/restart the printout, skip unwanted sections, single-step one entry at a time, etc. Using these controls, the user may intervene during the listing to avoid excess output, reset printer top-of-form between sections, or merely slow down a 1000 char/sec CRT display to a readable line-at-a-time.

HOW TO USE ZBIG

Under the North Star DOS, type the command: GO ZBIG program, followed by a carriage return, where 'program' is the name of the file containing the program you wish to analyse. Under CP/M, type the same command but without the 'GO' prefix. ZBIG aborts if the file is not found or is of the wrong type (type 2 under North Star DOS, type .ZBA under CP/M), then requests the output device with the question: "HARD COPY?". Responding with 'N' (for NO) causes output to the CRT console, while a 'Y' response (for YES) provides a printer listing of the same thing. When you specifying CP/M files, do not supply the file type (.ZBA) as part of the file name, as ZBIG appends it automatically.

You can optionally cause ZBIG to selectively display only those entries which have references to them in a restricted line range by appending the line number range to the start-up command, for example:

GO ZBIG FILENAME 1500 3600	Restricts listing to entries refered to in lines 1500 to 3600.
GO ZBIG FILENAME 2200	Restricts ZBIG to lines 2200 to the end of the program.

The entire program is still searched, but only entries containing line numbers within the selected line range are displayed.

At this point the listing begins and the following print-controls may be used:

SPACE-BAR: Freezes output prior to printing the next item.

ESCape: Skips all remaining items in the current section and proceeds to the next.

TAB (CTRL-I): Directs ZBIG to freeze the listing prior to printing the next section, and ring bell (CTRL-G) to inform the operator when there.

Once the listing freezes, the following additional controls may be used:

LINE-FEED: Prints the next item for each one typed.

CTRL-C: Aborts further processing, returning control to the DOS.

ESCape: Skips to the next section, but doesn't re-start the listing.

TAB (CTRL-I): Same as the TAB operation above.

Any other char: Resumes the listing printout from the current point.

Many variations are possible: omission of all controls allows output to proceed normally without interruption; repeated use of the SPACE-BAR will switch output on & off; once frozen, repeated line-feeds print labels one at a time. Typing the TAB right after the device# request causes the listing to pause just prior to the first section. These controls do not echo back to the terminal and affect output only as described.

ZBIG relies on the standard CTRL-C detect routine for reading the controls, and looks for them only between printed items. When more than one is typed during the previous line list, only the last one can be processed. But during a pause any number of controls may be used.

ZBIG CROSS-REFERENCE SAMPLE NORTH STAR RUN:

```
*GO ZBIG SAMPLE,3          <--- You enter your file name immediately
                                after the program name.

HARD COPY? N               <--- Enter 'Y' for console listing,
                                'N' for printer listing.

*** SCALAR FUNCTIONS ***

FND0():   485  2805
FND1():  5005  8070
FNF():   4405  4915  4930  7450

FNF1():   370  3610
FNG0():   470  1012  3810  7510  7520

FNL0():  1055  2240  5210  5310  7085  7150  7250  7320  7330  7340  7345
         7350  7400  7405  7465  7480  7615  7620  7725  8125
```

*** STRING FUNCTIONS ***

FNB0$: 355 2603 2608 2618 2655 2665 2915 3425

FNE$: 1038 2005 2010 2012 2045 2046 2075 2155 2512 2712 2715
       2745 2835 2920 3210 3220 3225 3320 3425 3715 3745 3845
       4860 5465 7760 8110

FNX$: 460 485 1012 3305 4710 4725 7510

*** ARRAY VARIABLES ***

N(): 115 3620 3625 3630 3635 3640 3642 4820 4850 4890 4915 4940
      4945

O(): 115 4050 4095 4115 4135

*** STRING VARIABLES ***

B$: 110 185 190 1125 2620 2625 2630 2932 2935 2945

C1$: 3105 3108 3110

H$: 110 5403 5405 5455 5460

N$: 110 330 335 355 365 2504 2655 2660 2665 2915 2925 2945
     3410 3415 3416 3865 4755

*** SCALAR VARIABLES ***

C2: 7055 7125 7135 8051 8070 8220 8250 8312 8325 8405 8410 8415
     8430

G9: 120 700 4880 4917 4925 7460 7475

L3: 3725 3735 3770 3772 3773 3774 4825 4845 4850 7075 7085 7105
     7115 7125 7135 7150 7250 7260 7400 7420 7525 7530 7610 7615
     7620 7725 7840 7845 7850 7930 7935 8060 8080 8240 8250 8320
     8325 8420 8430 8520 8530

Q: 3205 3220 3230 4903 4912 4915 4917 4925 4932 4935 4945 4948
    4950 4965 5405

T1: 7010 7030 7050 7105 7115 7125 7135

W2: 7315 7320 7335 7340 7350 7380 7390 7392 7420 7427 7450 7460
     7475 8050 8051 8070

Y: 8525 8530

```
*** GOSUB REFERENCES ***

GOSUB  1100:  2715  2835  3845  4715  4735

GOSUB  2200:   470   500
GOSUB  2230:  7615
GOSUB  2300:   325   550  2530  3420

GOSUB  2400:  2320  2820  4030  4075  7160  7435  7440

GOSUB  2900:   305
GOSUB  3400:   295
GOSUB  4800:   710
GOSUB  7000:   610
GOSUB  8100:  7230  7810  7910  8010  8210  8315  8418  8515

GOSUB  8300:   663
GOSUB  8500:   658


*** MISCELLANEOUS LINE# REFERENCES ***

  185:   265   440   600   685

  215:   205
  265:   275   278   285   295   305   315   340   350   355   365   370
         375   385

  440:   450   465   472   490   500   505   515   525   535

  450:   410
  552:   545   547   553

  600:   610   620   630   640   650   653   658   660   663
  685:   690   700   710

 2740:  2715
 2955:  2920  2925  2945

 3225:  3205  3220
 4155:  4040  4070
 4725:  4715  4732  4735

 4865:  4815  4855
 5010:  5010  5015
 7145:  7105  7115  7125  7135

 7155:  7145
 7160:  7040  7060  7075

 7262:  7258
 7430:  7365  7375  7385  7392  7400

 7940:  7905  7910  7930
 8075:  8070
 8515:  8540
```

8.1

-- 8.2 APCBASIC PROGRAM CRUNCHER --

A surprising amount of memory space is taken up by blanks inserted into the code and REMarks that have nothing to do with program execution. Well commented, structured BASIC programs have 30% or more of their memory area invested in blanks and remarks. The CRUNCH program conveniently optimizes a BASIC pgm by creating a new BASIC pgm without spaces and remarks, leaving the execution properties unchanged. This utility is useful only when more memory or program security is desired.

APCBASIC performs this CRUNCHing process automatically when programs are run from the operating system level (eg. GO APCBASIC program) and when LINKing (CHAINing) to a new program. This utility program is included so that convenient storage of a CRUNCHed program on a disk file is possible for program secrecy or for extremely large programs that cannot even be LOADed without size reduction. CRUNCH is able to reduce programs about 10% further than the automatic method used by APCBASIC, which must reduce the program prior to execution too rapidly to optimize the reduction.

-- FEATURES --

- Helps keep a program secret by removing all traces of internal program commenting and readable formatting.
- Allows programs with several thousand statements to be executed in 48K systems.
- Reduces program memory & file requirements by 20%-60% in only seconds, due to high-speed Z80 machine code. Such programs load faster and execute slightly faster. The memory saved increases the capacity for program variables and working storage.
- File-to-file conversion allows preservation of the original version.
- Deletes all REMarks from your program. Line number references to deleted REM lines (such as GOTOs, GOSUBs, or ERRSETs) are adjusted to the nearest non-REMark following the deleted lines.
- Spaces and line-feeds within quotes ("") are preserved; all others are deleted.
- Adjacent lines are joined together where possible to eliminate unnecessary line numbers (saving 3 bytes each). Resulting program lines may extend up to 200 characters. Line number logic is unchanged.
- CRUNCH aborts if the source program has any unresolved line number references. It displays the number of such occurences.
- Single version handles any APCBASIC or North Star BASIC program, including single & double density. The input & output files may be of independent density format. The CP/M version is identical in all respects.
- Prints the program size in bytes before & after the reduction process.

HOW TO USE CRUNCH

Under North Star DOS, type the command: GO CRUNCH source destination, followed with a carriage return, where 'source' is the original file to be CRUNCHed and 'destination' is the file that receives the reduced program. CRUNCH aborts the process if either file is of the wrong type (type 2 under North Star DOS, type .ZBA under CP/M). When you specify CP/M files, do not supply the file type as part of the file name, as CRUNCH appends it automatically.

CRUNCH requests confirmation of the destination file with the question "OLD FILE, OK?" or "NEW FILE, OK?", depending on whether the file already exists or not. Responding with a 'N' (for NO) immediately aborts the process; responding with a 'Y' (for YES) permits the process to continue: reducing the program, creating a new file if specified, then saving the reduced program on the destination file.

CRUNCH will request whether or not you wish to join lines where possible. You should respond 'Y' to have this done and 'N' to prevent it from being done. This option can reduce your program by another 10% as compared with not joining lines. CAUTION! Do not join lines in programs that contain MERGE or DELETE statements. These statements may be relying on unreferenced line numbers that the join option can potentially remove, causing the resulting program to execute differently than intended.

-- 8.3 APCBASIC CONFIGURATION PROGRAM --

Section 7.3 describes all the user-changeable options available in APCBASIC. The CONFIG program, written in APCBASIC, is designed to implement any combination of these options desired by the user in up to 4 versions of APCBASIC simultaneously. Before running CONFIG, read Sections 7.3 and 8.3 thoroughly and work out the changes you desire to make. CONFIG operates on the files containing the APCBASIC versions, rather than on memory copies, so all versions to revise must be on diskettes installed in your machine. Assumming RUN and CONFIG are on drive #1, run the program with the command:

```
GO RUN CONFIG        (Under the North Star DOS)
RUN CONFIG           (Under CP/M)
```

CONFIG is self-descriptive and executes in a conversational manner. Simply answer the questions CONFIG presents to you by typing the options you have previously worked out before running the program. CONFIG only implements options where they apply and can determine whether or not each option applies by examining the contents of the files. For example, if you change the Floating Point Board address, it bypasses non-FPB versions that you may also be configuring. Be sure to only modify copies of your original software while protecting your originals by keeping them safely away from your computer system as much as possible.

Always use a version not dependent on the floating point board for CONFIG execution. The FPB versions may require CONFIGuration for your system before they operate correctly (ie. if your FPB is addressed at other than EFF0 Hex).

**** 9.0 APCBASIC FOR NORTH STAR BASIC USERS ****

This explanation is designed for those people already familiar with North Star BASIC who require an understanding of the super-set facilities in APCBASIC. The APCBASIC features explained here are not in North Star BASIC. All other features not described are identical in both BASICs. Although the list of incompatibilities appears formidable, most programs require no change and nearly all the rest may be converted in one sitting. If you need more fundamental information on using APCBASIC, read the general instructions found in the other sections.

-- 9.1 COMPATIBILITY ISSUES --

Transparent Execution Enhancements

- A APCBASIC program may be run directly from the DOS by typing its file name immediately after the GO APCBASIC command in the following form: GO APCBASIC filename (followed by a carriage return).

- A special version of APCBASIC exists that RUNs a program, but all program development features have been removed to provide extra memory space for execution. This version, called RUN, has no command level where it says 'READY'. To run a APCBASIC program, you type 'GO RUN PROGRAM' in the DOS. This technique eliminates the need for the cumbersome AUTO-START method of running programs from the DOS provided in standard North Star BASIC. When the program ends, this version returns to the DOS level. In addition to more memory, RUN executes programs about 50% faster than the development version.

  After a programmed or CTRL-C STOP, RUN returns directly to the DOS. To CONTinue execution, simply do a DOS JP command to the restart location at 14 Hex after the load address of RUN. For standard DOS5.2 versions, the command is: JP 0E14 (followed by a carriage return).

- When a program is loaded for immediate execution, as in LINKing or load-and-go from the DOS level, all REMarks and extra spaces are removed from the program to provide maximum memory space for its execution. A slight speed improvement may also be accrued. When such an execution terminates, the program remains without spaces and REMarks, and must not be SAVEd over the original file version if you value your program comments. This process never occurs during program development, only on automatic program startup.

- Due to a different design approach in APCBASIC processes, most programs run from 2 to 5 times faster than under North Star BASIC. This increase will of course vary, and depends largely on the program being run. When the processing is bound by file or other I/O operations, only small speed gains are possible.

Potential Incompatibilities with North Star BASIC

- Generally, enhancements to APCBASIC have been incorporated to areas that have heretofore been considered errors in standard BASIC. Programs that rely on such errors, so that ERRSET recovery techniques can switch to alternate routines, might run into difficulty since such 'errors' no longer exist.

- All ERRSETs must be examined for compatability. In particular, calls to GOSUBs or FNs that setup ERRSETs for the program will not work, as the scope of each ERRSET is confined to the execution of the invoking sub-program. See details on the ERRSET statement in Section 5.4.

- The DATA-READ pointer is preserved during GOSUB and FN calls. If the subroutine itself alters the READ pointer for its purposes, this mechanism conveniently localizes it until a normal RETURN is processed. Thus GOSUB calls expected to revise the READ pointer before returning will not work.

- Token definitions since Release 5.0 will usually be treated differently in standard BASIC than in APCBASIC. As of Release 5.2, only three examples are known: LET, FILEPTR(I) and FILESIZE(I). North Star BASICs' LET token lists as WHILE and must be deleted, FILEPTR ends up as APCBASICs' MOD function, and FILESIZE becomes the APCBASIC SWAP statement. Simply change all MODs -> FILEPOS and all SWAPs -> FILESIZE in North Star BASIC programs to convert them to APCBASIC. Use the APCBASIC CHANGE command to rapidly find and update such program lines.

- MEMSET does not exist and in fact is unnecessary for all installations. The distributed copy determines the top of memory (non-destructively) automatically when loaded. The search uses the highest read/write RAM location supported under North Star DOS systems and under CP/M it uses the standard system vector at location 6 in memory for memory sizing. When this is undesired, use the CONFIG program described in Section 8.3 to set absolute limits on memory use.

- The LINE statement does not exist since APCBASIC generates no <CR> or checks for line length on output. But in APCBASIC this token LISTs erroneously as LOCAL and such statements must be deleted from the program. All inputs may be up to 160 characters.

- Your program REMarks may have strange spelling errors due to the keyword differences in APCBASIC. The quickest and easiest way to fix these is with the command: EDIT 0,REM which extracts all REMarks for your editing (see Section 2.2). The following list contains the changes you are most likely to require: DELETE->DEL, ENTER->AUTO, MERGE->APPEND, RENAME->REN, DOS->BYE, DIR->CAT, BY->STEP, LOCAL->LINE, DELETE->CLEAR, DELETE->SCR, LINK->CHAIN, POS->PTR.

- Do not attempt any of the North Star BASIC Personalization procedures on APCBASIC as they are not the same. Instead, use the CONFIG program provided since it implements all personalization options available under APCBASIC and in a much more straight forward manner. See Section 7.3 for details.

- Format specifications that include a dollar sign ($) may appear to APCBASIC as dynamic formats, which will subsequently execute incorrectly. Such formats appear to begin with a string variable (eg. Z$12F2, C$8I, etc). You can fix this problem by surrounding each such format with quotes ("), for example: %"Z$12F2", %"C$8I", etc.

Reserved Words and Special Characters

APCBASIC uses a number of keywords not found in North Star BASIC. These are shown with their equivalent in each language. For convenience to the North Star user both are accepted from the keyboard by APCBASIC. However only the APCBASIC version is printed in all program listings. Note that old keywords in existing programs will not have to be changed. APCBASIC will automatically translates them to their proper names where ever required.

APCBASIC	NORTH STAR BASIC	PURPOSE
ENTER	AUTO	Numbering Input Lines
DIR	CAT	Disk directory listings
SIZE	PSIZE	Program size
DOS	BYE	Exit from BASIC to DOS
SAVE	NSAVE	Loading APCBASIC programs
BY	STEP	FOR..NEXT loop step size
NOMARK	NOENDMARK	End-of-file mark control
INCHR$	INCHAR$	Single-character inputs
LN, LOGE	LOG	Logarithm base(e)
LINK	CHAIN	Automatic program sequencing
FILEPOS	FILEPTR	File position function
;	\	Statement separator
CTRL-R	@	Editing control character

Other Minor Syntax Differences

- DEL SCRatches entire program when no args are given, but user verification is required (Y or N response). SCR will DELete line ranges if followed by the line range (DEL and SCR are equivalent). When specifying a line range that includes the last line in the program, a dollar sign ($) may be used to signify that last line (eg. DEL 1,$). See Section 5.7 to see how to execute this command from within a program.

- FN DEFinitions must appear as the 1st statement on the line that you define them. North Star BASIC allows it to appear anywhere in the line which results in more time spent in the load-up process prior to program execution.

- Through system errors in some North Star BASICs, program lines may contain control characters - erroneously. APCBASIC will display such bad characters as question marks (?) for your correction.

- Lines, commands, and direct statements may be entered in any combination of upper and lower case. APCBASIC converts all lower case letters not inside quotes ("") to upper case before proceeding.

- User defined names with more than one character (eg., T1, A$, FNS6, Z3$, etc.) must be entered run together without any inserted spaces to avoid a syntax error. The left parentheses following function, string, or array names is considered part of the name when applying this rule.

- Line-feeds may be entered into the program text for longer lines, and so that spaces can be inserted between lines to make the text more readable. When LISTed, each line-feed expands into a LF-CR sequence.

9.0

-- 9.2 PROGRAM DEVELOPMENT FACILITIES --

All the commands in North Star BASIC are supported, but in a much expanded form. Only brief references to their extended capabilities are provided below and you should read Section 2.0 for all details concerning their use.

Program Entry, Storage and Retrieval (Section 2.1)

- Programs can be entered into APCBASIC from ordinary text files. Such a file must contain the same sequence of characters that you would normally type into APCBASIC through the console keyboard. This facilitates conversions of programs written in other BASICs to the APCBASIC environment.

- You can tell the LIST command to display only those lines which contain a specified search string. This greatly simplifies the job of locating items of interest within your program. The North Star paging method is not supported. You can control the LISTings by touching the space bar to start and stop the listing. Once stopped, a line-feed (CTRL-J) or a carriage return (CTRL-M) produces lines one-at-a-time.

- The APPEND command has been replaced by the MERGE command, which provides a generalized facility for adding APCBASIC code lines from other files. Line ranges in both memory and source file are unrestricted. On line# conflicts, MERGE replaces the old line with the new, while the remaining lines are inserted into their proper places.

- APCBASIC prints the program size automatically after a program MERGE, LOAD, or SAVE. SAVE will save a program on a new or existing file. In either case APCBASIC will respond with an OLD or NEW FILE message and request user confirmation of the SAVE (Y/N response). NSAVE is identical with SAVE in APCBASIC and a file length specification may follow a new file name. If you type SAVE without a file name, APCBASIC uses the file name specified by the last SAVE command accepted. This feature makes program backup during development fast, repeatable and mistake proof.

Program Editing and Alteration (Section 2.2)

- The EDIT command displays the line being editted before you begin your changes. On completion of each line, it proceeds to successive lines as long as you desire. You can skip lines without editing them and proceed to the next automatically. EDIT can also select lines for you to edit which contain some specified string while skipping over the rest. See Section 1.6 for the summary of editing control keys available.

- A global search-and-replace facility is provided by the CHANGE command. This can replace one string with another everywhere or selectively within a line range. APCBASIC will request 'VERIFY?' to allow user control of each replacement ('Y' response). An 'N' response causes replacement of all occurences found.

- The RENumbering facility can be restricted to affect only a subrange of lines within a program. It also supports rearrangement of groups of lines and can be made to renumber a program without disturbing the increments between lines.

Execution Control and Debugging (Section 2.3)

- Direct statements are not limited to only one statement. Any correctly formed multi-statement line can be executed, including FOR..NEXT loops and IF..THEN..ELSEs. GOSUBs can be called and FNs may be employed in string or numeric expressions (after a STOP or END). GOTOs and RETURNs may be used to CONTinue program execution after a STOP.

- The TRACE command provides the ability to 'walk' through your program during execution and watch each statement as it is executed. Break points may be set and various keyboard controls provide a testing 'harness' over your program that keeps you in control at all times. Conditional break points allow your program to execute normally until some condition is met, at which time execution breaks and debugging facilities come into play.

-- 9.3 PROGRAM CONTROL ENHANCEMENTS --

- The DOS (or BYE) command may be used as an executable statement to exit into the DOS directly, instead of back to APCBASIC. (Section 5.4)

- Computed GOSUBs can be done with the ON <expression> GOSUB <line# list> statement that works exactly like the ON... GOTO... statement, except that control passes to the following the <line# list> on RETURN from the GOSUB. See Section 5.4 for further details.

- ERRSET statements are confined to the scope the current subprogram level in which they are defined. It is possible (and highly useful) to setup different ERRSET traps at each subprogram level at the same time. This approach usually supports ERRSET traps in North Star BASIC programs, but you should validate this when bringing up such a program for the first time under APCBASIC. This feature should be well understood to avoid potential incompatibilities.

- The IF statement supports multi-level IF..THEN..ELSE structures and any THEN or ELSE clause may be a compound statement consisting of one or more statements if needed. The conditional expression may contain any combination of string and/or numeric comparisions, expressions and logical operators. Eg. IF A$=B$ AND NOT (Z=Y OR C$>"XYZ") XOR W>X...

- The EXIT <line#> statement will now jump to the statement following the currently active FOR...NEXT loop, if the line# is omitted. This allows more convenient use of the EXIT statement.

- User-DEFined functions may be DEFined with no arguments when this is desired. Both multi-line and one-liners are supported. (Section 3.5)

- Program files may be MERGEd during execution in the same way as described in Section 2.2. Sophisticated overlay structures can be created using this feature. For instance, your 'core' routines can always stay in memory while special purpose libraries can be MERGEd into the program as they are needed. Use the same syntax as the MERGE command. (Section 5.7)

- To remove unneeded programming after it is executed (such as one-time initialization routines), use the DEL statement in the running program just like the DEL command. All variables and open files are preserved. See Section 5.7 for important details.

-- 9.4 DATA MANIPULATION, CONTROL and CONVERSION --

- Variables may be passed between CHAINed programs by listing them in the CHAIN statement after the program file name expression. To preserve all variables, use an at-sign (@) instead of the variable list.

- Local variables may be defined using the LOCAL statement within GOSUBs and FNs. These variables may be used for any purpose without fear of over-writing the values of variables of the same name outside the subprogram. Use within recursive procedures makes this is a particularly useful and powerful tool. See Section 5.5 for further details.

- Variables may be re-DIMensioned at any time. Unneeded memory space is returned for re-use. A dimension error is never generated for this. See Sections 3.3, 4.4, 4.5 and 5.1 for more details.

- The initial values stored in string and numeric variables and arrays when created can be quickly restored using the RESTORE statement followed by the list of variables. (Section 5.1)

- Conversions of string values to floating point will accept leading as well as trailing spaces but non-numeric characters found elsewhere in the string will generate a trappable VALUE error.

- The DATA statement READ pointer is localized within GOSUBs and user-defined functions (DEF FN...) so that multiple-nested DATA lists may be used without conflict. (Section 5.1)

- A multi-case ON..RESTORE statement permits a computed selection of the next DATA statement to be accessed by the READ pointer. (Section 5.1)

- Random numbers are used the same way as in North Star BASIC. However the generator is based on random 24-bit values in APCBASIC instead of 16-bit. This yields much longer random sequences, and better distribution over small intervals.

- Logical expressions may be used to generate a floating point zero or one where ever a numeric expression is expected. Any complexity of logical connectors, comparisons, or data types may be used as long as comparisons involve like data types. Strings, values, arrays, and bits may be combined within a single expression, for example. (Sections 3.4 and 4.4)

- Exchanging string or numeric values between variables is supported by the SWAP and SWAPDEF statements, which execute 3-100 times faster than assignment statement implementations. (Section 5.3)

- The LEN() and ASC() functions both accept general string expression arguments in addition to string variables. The ASC() will return -1 if its argument is a null string, instead of an OUT OF BOUNDS ERROR. (Section 6.3)

- Positive integers 1 to 16 bits wide may be packed into (or unpacked from) string variables with the BIT function (Section 6.3).

- MIN() and MAX() find the minimum and maximum values among a list of numeric values. (Section 6.1)

- The MOD(N,M) function returns the smallest positive value V such that N-V is divisible by M. This is different than the 'straight' remainder when negative args are supplied.

- ASIN(X) and ACOS(X) are available to return the arcsin and arccos of expr X. In both cases the input expression must be values between -1 and 1.

- The CEIL(X) function, similar to INT(X), returns the 1st integer greater than or equal to X. It is equivalent to -INT(-X).

- The TRUNC(X) function returns X without a fractional part. Note the difference between TRUNC(X) and INT(X). This is equivalent to the expression SGN(X)*INT(ABS(X)).

- The SGN(X) may be used with two arguments, SGN(X,Y), to return the value of Y with the sign of X. [Eg. SGN(-3,34)=-34, SGN(0,45)=45, etc.]

- FRAC(X) returns the fractional portion of X. (Eg. FRAC(45.19) = 0.19). This is equivalent to X-INT(X). Note that X may also be an expression. Note also what occurs given a negative expression: FRAC(-305.7) = 0.3

- PI is a function without arguments that returns the value 3.1415926535898 rounded to the prevailing precision of APCBASIC.

- ROUND(X) returns X rounded to the nearest integer. ROUND(X,P) rounds X to a maximum precision of P digits, where INT(P)>0.

- TAN(X) returns the tangent of X given as an angle in radians.

- XOR (exclusive OR) and EQV (equivalence) have been added to the numeric logical operator repertoire. They fall after AND and OR in the operator precedence and EQV has the lowest of all operators.

- Polynomials from degree 1 to 255 may be evaluated using the POLY(X,A(),D) function described in Section 6.2.

- Powers (eg. X^Y) of negative numbers are permissible if the exponent is an integer from -9999 to 9999. For example: -2^3 = -8. X^0 = 1 for any X. See all the numeric operators in Section 3.4.

-- 9.5 DEVICE I/O AND EDITING --

DEVICE I/O ENHANCEMENTS:

- The POS() function returns the current column position of device 0 to 15.
- The LINES() function returns the current line position of device 0 to 15.
- Multiple blank lines may be generated from a single PRINT statement by a field of slashes, similar to FORTRAN format statements. For example: PRINT #D,///, will generate 3 carriage returns on device D. Slashes may be interspersed throughout a PRINT statement where ever needed.
- Dynamic formatting is achieved by following the format character (%) with a string expression which evaluate to a legal format specification.
- INPUT statements permit specifying separate prompts for each variable to be input. Each prompt may be specified with a string expression instead of being limited simple quoted strings. (Section 5.2)
- Re-direction of Device I/O to and from files is made possible using 'devices' 8 thru 15, which are assumed to be file numbers previously assigned with the OPEN statement. This permits access to files as if they were I/O devices in commands (ENTER, LIST), statements (INPUT, PRINT), and functions (POS, LINES, INCHR$) that use #device numbers. See Section 5.2 for details.
- Formatting in PRINT statements is considerably extended. See Section 5.2 for all the details.
- The default I/O device (always 0 under North Star) can be altered using PARAM(3), described in Section 5.8.

ENHANCEMENTS TO THE LINE EDITOR:

- Line editing now has a CTRL-X control that deletes all characters up to but not including a char typed after the CTRL-X. This combines the actions taken by the CTRL-D and CTRL-Z controls.
- CTRL-F performs the combined effect of CTRL-G, CTRL-N, CTRL-G.
- The EDIT command may be executed within a program to allow editing a string not previously entered. Use the syntax: EDIT <string expression>
  This statement will place the string expr evaluated into the old-line buffer so that editing CTRL characters may be used on it. Very powerful tool for editing data of many kinds, not just keyboard entries.
- See Sections 1.4, 1.5 and 1.6 for full details on the APCBASIC line editor.

- String indexing is not limited to string variables. Any string constant, variable, function, subexpression or indexed string may be followed with an indexing expression. The six different indexing modes include the two two in North Star BASIC. Instead of generating OUT OF BOUNDS ERRORS, APCBASIC accesses only the real string portion within the indexed area. See Section 4.5 for a thorough explaination of string index.

- String expressions support string repetition (* operator), bit-string operations via the boolean operators (AND, OR, NOT, XOR and EQV), and operations are controlled by precedence or parentheses. See Section 4.4 for the full discussion.

- The STR$ function may have an optional 2nd string expression argument to specify the format for converting the value.

- Functions MATCH() and FIND() provide high-speed general purpose string searching mechanisms which have many useful applications (eg. table lookup, sorting and implementing case statements).

- REV$(string exprn>) returns a reversal of the string argument supplied.

- TRIM$(<string exprn>) returns the given argument stripped of lead/trailing spaces.

- Strings of any length may be rotated left or right by 0 to 255 bit positions at once using the ROTAT$(S$,R) string function described in Section 6.3.

- CHR$() function may have two arguments to produce an ascending sequence of ASCII codes, instead of just one code (see Section 6.3).

- The TRAN$(A$,B$,C$) function translates character codes of A$ found also in B$ into their corresponding characters in C$ (identical to the TRANSLATE() function in PL/1). See Section 6.3 for further details.

-- 9.7 FILE PROCESSING ENHANCEMENTS --

Detailed information about file processing enhancements can be found in Section 5.6 for statements, or Section 6.4 for functions.

- A file may be OPENed under several different file numbers simultaneously. Certain applications execute 5-100 times faster with several independent buffers are carefully used to process a single file (eg. file sorting and compaction).

- 16-BIT values may be READ/WRITTEN to file by placing an at-sign (@) prior to the variable/expression in the I/O list. This is similar to the 8-BIT data file I/O that uses the ampersend (&) for its designation.

- The ampersand (&) can direct 8-bit data transfer between files and strings, in addition to numbers. For example the statement READ #1,&X,&A$(I,J) reads one byte into X, and J-I+1 bytes into A$ starting at position I. Note that the number of bytes read (or written) is length of the string (or string expression) specified.

- Up to 16 files may be opened simultaneously instead of just 8. Note that each open file requires 512 bytes (256 in single-density systems) of memory for its own private use. With 16 files open, this would consume 8192 bytes that cannot be used for either the program or its data. Always try to maintain the least number of open files at any one time.

- The random file positioning expression in READ and WRITE statements may be used anywhere in the data lists of those statements. Thus the percent sign (%) need not immediately follow the file number if not desired. To position a file without performing actual file operations, simply state a file operation without data. For example: READ #2 %P1 or WRITE #F %P2.

- The end-of-file mark written after each WRITE operation may be suppressed for all subsequent WRITE operations using the statement: NOMARK <expr>. A non-zero expression causes file mark suppression, and a zero expression brings it back again. Most useful for random or binary file processing.

- The RENAME statement performs the obvious transformation on a file name.

- The FREE statement (not the function) releases all dead space left by files after they are CLOSEd. This statement must NEVER be used while a FOR..NEXT loop or FN is in progress -- it may cause a APCBASIC system crash. See Section 5.6 for details.

- DIR (CAT) is also an executable statement.

- CLOSE will close all open files if no file number is specified.

- SPACE(D) returns the number of blocks (256 byte blocks) remaining on disk drive D. If D=0 then the default drive is assumed.

- The FILESIZE(expression) function returns the number of blocks in any open file as specified by the expression. The FILEPOS(expression) function returns the file-position of the OPEN file specified.

- The default disk drive (always 1 under North Star) may be altered with PARAM(2), described in Section 5.8.

-- 9.8 SYSTEM INTERFACE --

The various North Star machine level interface capabilities are supported within a much expanded framework and access to certain execution properties internal to APCBASIC is also made available. See Section 5.8 for system interface statements, Section 6.5 for system functions.

- The FILL statement permits a list of values instead of a single value. Strings may be FILLed into memory and numeric values may also be FILLed in 16-bit format, in addition to the 8-bit format.

- The EXAM function still exists, but an EXAM statement is provided to support the inverse capabilities of the FILL statement mentioned above.

- The CALL function may have one or more arguments. The last argument is sent through the DE register, and all preceding arguments are saved on the stack in reverse order (except the first argument, the jump addr). A CALL statement is also provided (unrelated to the function) that supports full access to all the CPU registers.

- Using brackets [] around any variable or array element evaluates to the address in memory of that variable. Useful in the CALL function to pass pointer arguments, and in FILL or EXAM statements to directly access the contents of program variable memory cells.

- The FREE(expr) function returns the number of unused memory bytes available at any point in time by specifying any positive expression. It returns the starting memory address of that area if you give a negative expression. This area may be used for scratch work provided that no new variables or file buffers are created during its use.

- The PARAM(P) statement allows control of several internal execution factors. It may be used on the left side of an assignment statement (=) to assign new values, or on the right side to determine their current setting. See Sections 5.8 and 6.5 for details.

- The INP() function returns a character or a number, depending on the type of expression it is used in.

- The OUT statement will accept either a string or numeric expressions for the byte value to be OUTput. When a string is supplied, only the first character is sent.