SBC Reference Manual

SHARP PC-1270

BASIC Compiler

SBC Compiler Description 3

Sharp BASIC Language 4
- Overview 4
- Arrays and Flexible Memory Management 5
- Compiler Directives 6
- Implied Multiplication 7
- Labels 9
- Line-Continuation Symbols 10
- Maximum Program Sizes 11
- Passwords & Copy Protection 12
- REM Statement and ' Comment Symbol 12
- Legal Symbols 14

Compiling a Program 15
- Command Line Switch Options 15
- How SBC Compiles Programs with No Line Numbers 16
- Removing the Line Numbers from a Source File 17
- Sample Source Listing 18

Decompiling a Program 20
- Sample Decompiled Program 20

S'BASIC Statements & Functions 21

Error & Warning Messages 30
- Fatal Compiler Errors 30
- Non-fatal Compiler Errors 31
- Compiler Warning Messages 36
- PC-1270 Run-Time Errors 38

Utilities 39
- BANKS.EXE Linker for Multi-Bank Flash Card Programs 39
- COPYLST.EXE 41
- RENLST.EXE 41
- Transferring Programs to Cassette Tapes and Other Calculators 42
- Entering Alpha Characters in the PC-1270 43

South [1]Burlington VT 05403-6275 USA · 855 255-8080· FAX 802 862 8357

1

SBC Compiler Description

SBC™ is an BASIC compiler which converts a standard ASCII text file into the internal object file format used by the Sharp PC-1270 and other Sharp Calculators.

In addition to the features included in the Sharp PC-1270, the SBC compiler provides many other features which enable the programmer to write and maintain programs in considerably less time with greater accuracy and readability. Features include:

- No line numbers are required in the source file, virtually eliminating the problems of running out of line numbers and "bunched-up" line numbers. **GOTO**, **GOSUB**, and **RESTORE** statements reference labels in the source file. The labels are eliminated during the compilation process so they take no space in the finished program.

- Blocks of code can be moved freely around a source file, or from one source file to another, without any concern about line numbers. This makes it quite easy to optimize and/or reorganize a program without having to repeat the debugging process.

- Programs can be edited on a PC with any standard ASCII text or programmer's editor such as MS-DOS™ EDIT, or BRIEF™. This provides all the advantages of a full-screen editor with search and replace and other features.

- Virtually any number of statements can be concatenated into one BASIC line through the use of two line-continuation symbols saving space in the target calculator. Source-file readability is greatly improved because conventional notation can be used in **FOR..NEXT** loops and **IF..THEN** constructions in the source file without taking any additional space in the target card or calculator. Eliminating 100 BASIC line numbers from a program will reduce the compiled program by 200 bytes (which may be enough to fit the program into a smaller and less-expensive card).

- White space and comments can be used freely within the source code to make the code readable and fully documented. Both are stripped from the compiled code so no space is consumed in the finished program.

- Documentation can be included right in the source file, where it belongs, without adding any space to the compiled program. The documentation is also secure because it never goes into the target card or calculator.

- SBC will compile programs for the P*ROM CF Series of Flash cards for all card sizes including 8K, 16K, 32K, 64K, 128K, 256K and 512K. It will also compile programs for Sharp RAM cards, and for the Sharp PC-1248/PC-1250 series of calculators. All the Sharp S'BASIC statements and functions that can be executed programmatically are supported.

- Multiple source files can be compiled in one pass. For larger Flash cards with many files (2, 4, 8, 16, 32, or even up to 64), you can compile and link all the files for a project with a single command.

- Programs can be decompiled to generate the exact code the PC-1270 is executing.

- Flexible memory usage is automatically calculated and reported showing the arrays dimensioned and their space requirements.

Sharp BASIC Language

Overview

BASIC statements for the PC-1270 are fairly conventional and are very similar to the other Sharp pocket computer BASIC languages with the exception of line numbers. In this new version (Version 3) of the compiler, line numbers are no longer required in the source file. See "How SBC Compiles Programs with No Line Numbers."

Line numbers are automatically assigned by the compiler. Up to 999 BASIC lines can exist in a program bank. Labels are used as the destination of **GOTO**, **GOSUB**, and **RESTORE** statements in the same manner as used in many BASIC languages.

If a line number appears at the start of a line, it is used, the automatic assignment of line numbers is adjusted to proceed upwards from the stated value. This allows the compilation of numbered files and of "mixed" files where some lines are numbered and others are not (a situation which exists when files are converted from numbered to unnumbered).

If a line number is the same or less than the preceding line number, the calculator considers it the beginning of a new program and will stop processing automatically at the end of the program being executed. **GOTO** and **GOSUB** commands will only search for the indicated line number within the program in which they were executed and if not found, will terminate the program with an **ERROR 4**. The compiler will generate a warning if it detects a non-ascending line number.

A BASIC program line can be virtually any length, even thousands of bytes long. The line-continuation symbols **":"** and **"\"** can be used at the end of each line in the source file to put many lines of source code into a single BASIC program line. Use of these symbols enable you to reduce the number of BASIC program line numbers assigned. Therefore, once these modifications are made, it is very unlikely that you will ever run out of line numbers. Very complex programs for the PC-1270 rarely use more than 200 lines. An additional advantage is that each line number saved will save 2 bytes in program length. For more information about continued lines, see Use of the Line-Continuation Symbols.

In the source file, each line of text should be limited to the screen width of 80 characters for convenience and readability. See the Sample Source Listing.

If an **IF** condition is false in an **IF..(condition)..THEN** construction, processing stops at the **THEN** statement and does not resume until the start of the next BASIC line. (If the word **THEN** is not used, processing stops immediately after **(condition)**.)

If you attempt to use a Sharp PC-1250 series calculator to edit files created with SBC, you must be careful to have no BASIC program line longer than 80 bytes. Although longer lines will execute properly in both the PC-1270 and PC-1250 Series, the maximum length line the editor in the PC-1250 Series can handle is 80 bytes. If a longer line is brought into the editor, the line will be truncated to 80 bytes in length.

Except for blank lines and continued lines, each line must start with either a legal BASIC verb or the **(')** remark symbol. Blank lines contain only "white-space" followed by a carriage return; continued lines are a continuation of the previous line which ended with the line-continuation symbol "\" or **":"**.

The program in the PC-1270 is started by pushing one of the eight user-defined function keys on the PC-1270. These keys look for an S'BASIC label at the start of a program line and, if found, begin executing code at that point. The program stops executing when an **END** or

STOP is encountered, there are no more lines in the program, or the next BASIC line number is equal to or less than the current value.

Arrays and Flexible Memory Management

The Sharp PC-1270 calculator has one contiguous area of memory that is used both to store the BASIC program and all variables. (There is also a small area used by the calculator that is not available to the programmer.)

A Flash card provides 16K of memory to the PC-1270 at any one time. The first 8K is used to store program code, and the upper 8K to store variables. Thus there is always 8K of program code and another 8K of variable storage. To accommodate larger programs, you can have many 8K “banks” of program code (up to a total of 64 for a total of 512K).

In RAM cards, the program and data areas are shared. Therefore, the more variables (in the form of arrays) you use, the less memory is available for your program.

The section entitled Maximum Program Sizes describes the maximum length of a compiled program that will fit into the target card or calculator. The compiler will compute the length of the compiled program and warn you if it exceeds approximately 90% of the available memory. If the compiled program size exceeds available memory, an error is reported. If you use arrays, the size occupied by the arrays is automatically computed by the compiler.

Arrays are dimensioned with the **DIM** statement. They can be numeric (real numbers only) or string type and can have 1 or 2 dimensions. There can be up to 256 rows and 256 columns (although all memory would be exhausted by such an array).

Numeric Arrays

Each numeric array uses 8 bytes per element plus 6 bytes for overhead. For example the array **B(24)** has 25 elements (the first being 0) and will consume **25*8+6** or 206 bytes.

String Arrays

String-type arrays (except for the special case of A$, see below) have a default element length of 16 bytes, however, you can set it at any length you wish (from 1 up to 80 bytes [numbered from 1 through 80]). They can have one or two dimensions and up to 256 rows and columns (numbered from 0 through 255). The array **C$(11,2)*24** is a two-dimensional, 12-row, 3-column array of 24-byte strings and would consume **12*3*24+6** or 870 bytes. (The rows are numbered 0 through 11, the columns 0 through 2.)

With the exception of the letter "A", the same letter designator can be used for both a real and string array, e.g., you can have an array **E(23)** (24 real numbers), and the string array **E$(30)** (31 16-byte strings). Thus there are a total of 51 arrays that can be used.

The A() Array

The 26 single-letter variables A-Z (or A$-Z$) are allocated memory in the calculator whether or not you use them. These can also be addressed as elements of the A array, e.g., A(2) is the same as B. You can mix real numbers and strings within this A array, i.e., **A(1) = 1** and **A$(2) = "TWO"** are both legal (however, strings always have a maximum length of 7 bytes). The A array can be extended beyond the letter "Z" or 26. If you use an A (or A$) element greater than 26 (equivalent to Z or Z$), memory is allocated at 8 bytes per element plus overhead of 6 bytes. For example, if the highest element of the A array used is A(28), the array space required is 2*8+6 or 22 bytes.

You can dimension the A array by using the **DIM** statement (subject to limitations, see below) or by simply using an element greater than 26. For example, the statement **A(50) = 1500** has the same effect as the statements **DIM A(50) : A(50) = 1500**. There is no difference between the statements **DIM A(50)** and **DIM A$(50)**. The A array is unusual in that each element can be either a real number or a string (but not more than 7 bytes in length), and the element can be switched back and forth. The following sequence is legal:

```
A(50) = 50
A$(49) = "FORTY-NINE"
A$(50) = A$(49)
PRINT A$(50)
```

If we execute the above statements, we will get **"FORTY-N"** on the screen because the string length for the A array is always 7 bytes. You cannot specify the string length of the elements of the A array in the **DIM** statement. Another difference between the A array and other arrays is that the A array has no "0" element, i.e., if you try to execute **PRINT A(0)**, you will get run-time **ERROR 3**. The A array always has one dimension and cannot be dimensioned with less than 27 elements (the first 26 elements are the single-letter variables A - Z and are always present).

For Flash cards, the A() array must be dimensioned if used in the program. The compiler reports an error if it is used and not dimensioned.

For RAM cards, it is good practice but not required to dimension the A() array. If it is used and not dimensioned, a warning is reported and the computed array-size calculations will not include the effect of the A() array. The calculator will report run-time **ERROR 6** if it runs out of memory.

The only way to remove an array (i.e., free the memory used by an array) is to issue the **CLEAR** statement. Unfortunately, this removes all arrays and also clears the A-Z (or A$-Z$) values.

Refer to the P*ROM Application Note Programming CF Series Flash Cards for more information.

Compiler Directives

Compiler directives are instructions for the compiler. They generally must occur prior to the first BASIC program line or an error will be reported. Compiler directives can be in upper or lower case (or a combination).

#ARRAY_SIZE (size)

This directive is optional but can be used to tell the compiler the amount of memory to reserve for arrays in the target calculator. SBC automatically computes the space consumed by arrays and will report the total amount.

You can use the **#ARRAY_SIZE** directive to specify the amount of space used by arrays and override the value computed by the compiler. If a value is specified in the source file and it is different than the value computed by the compiler, a warning is issued. The value specified in the file is used, but unless you are absolutely sure it is correct, you should use the value computed by the compiler. (If an incorrect value is used, data in the arrays will be lost each time the calculator is turned off.)

The amount of space is entered in bytes, i.e., the directive **#ARRAY_SIZE 256** will reserve 256 bytes for arrays. See Arrays and Flexible Memory Management to determine how much array space is required. Only one **#ARRAY_SIZE** directive may appear in a source file.

In multi-bank Flash card programs, the array size should only be specified in the first bank (Bank 0). Values specified in other banks are ignored.

#CARD_SIZE (size)

This directive specifies the size Flash or RAM card, or target calculator for which the program is being compiled. Valid card sizes are:

15	8K Flash Card bank (use for all Flash card sizes)
2	2K RAM card (CE-210M)
4	4K RAM card (CE-211M)
8	8K RAM card (CE-212M)
16	16K RAM card (CE-2H16M)
3	PC-1250 (2K)
5	PC-1250A (4K)
9	PC-1251H (10.2K)

When compiling a program, its length is checked against the size card you have specified to see if the program will fit. If it is too large, an error will be reported at the line that caused the overflow (see also **#ARRAY_SIZE** below). If no **#CARD_SIZE** directive is found in the source file, the compiler will use the default card size specified in the command line, if any. If no **#CARD_SIZE** was specified in the source file and no default card size was specified, a card size of 15 is used. Only one **#CARD_SIZE** specification is permitted in a source file.

The length of the program is checked to see if it will fit within the specified calculator. The program is actually compiled using a 16K RAM card format to be transferred to the PC-1250 series calculator.

Programs can be compiled for other Sharp Pocket Calculators by specifying a **#CARD_SIZE** of 9. These can be transferred from a 16K RAM card to the target calculator in the same manner used for the PC-1250 series.

Implied Multiplication

The S'BASIC language permits the use of implied multiplication. When two (or more) numeric variables are found together without intervening operators, the variables are multiplied by each other even though there are no multiplication operators. For example, the statement **PRINT ABC** is equivalent to **PRINT A*B*C**.

The SBC compiler allows the programmer to specify the use of implied multiplication in two ways.

Virtual Operator "|"

The first method, which is greatly preferred, uses the **"|"** virtual operator to specify implied multiplication. This method is preferred because 1) it allows you to use any sequence of letters (even those that contain keywords), and 2) it allows the SBC compiler to maintain its high level of error detection.

To specify implied multiplication, we should write

PRINT A | B | Z | X

which is compiled to

PRINT ABZX

(The **"|"** symbol is found on U.S. keyboards as Shift "\" (backslash). It is the symbol used in the C language for the binary "or" operator and has the ASCII value of 124 (7CH). It is called a virtual operator because it does not actually take any space in the compiled code. It is just a way to tell the compiler that you specifically want implied multiplication to occur.)

The source code must contain a **"|"** operator in all places where you wish implied multiplication to appear or an error will be reported. You can use any sequence of letters, even if they form or contain a keyword. For example, the statement **PRINT LIEIN** will be compiled as **PRINT LEN** and "LEN" will be interpreted as **L*E*N**. (This is NOT true if the second method described below is used.)

Command-Line "-i" Option

The second method is to use the **"-i"** command-line option. When the **"-i"** option is set, the compiler accepts any sequence of letters (which does not contain a keyword) without intervening operators as a group all of which are multiplied by each other. Therefore it is impossible to detect undefined variable names or spelling errors. For example, the following line would normally be reported as an error:

PRINT ABZ X

because "ABZX" is an unknown keyword. If the implied multiplication option **"-i"** option is specified, however, the line will be compiled as the equivalent of

PRINT A*B*Z*X

Also, the code segment

PRINT LEN

would not produce the intended results because the "LEN" would be interpreted as the LEN() function (which returns the length of string), and not as the product of **L*E*N**.

It is for these reasons that the use of the **"|"** operator is preferred.

Implied Multiplication Rules

Implied multiplication can be used between any two numeric variables (simple or array), and between a leading constant and one or more variables, e.g., **12 * A * B** can be written as **12 | A | B.** Simple and array numeric variables can be used in any sequence, e.g., the following is legal

B = 1.2 | F(10) | A | B | S(20,2)

This is compiled as (note that the "|" operators do not take any space in the compiled code):

B=1.2F(10)ABS(20,2)

Note that the sequence ABS(20,2) is properly interpreted as A * B * S(20,2) because of the **"I"** operators.

Implied multiplication has the highest priority and is performed before power or division operations. It does not follow the normal rules of priority, for example, **B/CD** is not the same as **B/C*D**, but is the same as **B/(C*D)**.

If you use the **"-i"** command-line option method, be careful that the sequence of letters in an implied-multiplication group does not contain a keyword in the S'BASIC language, e.g., the segments **PRINT LEN** and **PRINT ALEN** would not give the intended results because the group of letters "LEN" would be compiled as the BASIC function LEN() which returns the length of a string. It is preferred to use the **"I"** virtual operator, e.g., writing **PRINT LIEIN** and **PRINT AILIEIN** will give the intended results of **PRINT L*E*N** and **PRINT A*L*E*N**.

Decompiling Programs Containing Implied Multiplication

When a program that contains implied multiplication is decompiled, the virtual operator **"I"** does not appear in the decompiled code. For example, if the original source file contains the phrase

```
PRINT AILIEIN
```

the decompiled code would show the phrase

```
PRINT ALEN
```

This cannot be recompiled accurately, even with the **"-i"** option, because of the embedded keyword "LEN". To recompile the code accurately, you would have to reinsert the **"I"** virtual operator symbol between each pair of letters.

Labels

There are two types of labels you can use within an S'BASIC program. The first is called an SBC Labe

SBC Labels

An SBC Label is placed at the left margin of the source file. The label must start with a letter and contain only letters, numbers or underscore **"_"** characters. It must be terminated with a **":"** with no intervening spaces between the last character and the **":".**

Only one label can be used for a line, and duplicate labels are not permitted. A label is "visible" only within the source code being compiled.

As in other keywords, labels are not case-sensitive. The labels **"Sum_totals"**, **"SUM_TOTALS"**, and **"SuM_toTaLs"** are all equivalent. From purely a style point of view, you may want to consider using a single capital letter followed by lowercase characters for labels as this makes them immediately identifiable as SBC Labels to a reader of the source code.

```
IF X = 0 THEN \
        GOSUB Sum_totals
PRINT "DONE"

Sum_totals:
    FOR Y = 1 TO 50 :
```

```
...
...
```

S'BASIC Labels

Sharp S'BASIC also allows one or two letters to appear at the start of a line as a label. This label can be addressed by **GOTO**, **GOSUB** and **RESTORE** commands in the conventional manner, e.g., **GOSUB "ST"**.

Labels must appear at the start of a program line and must be 1 or 2 letters embedded in quotes, e.g.,

```
"A" PRINT "THIS IS LABEL A"
"AB" PRINT "THIS IS LABEL AB"
```

When pressed, the user-defined function keys search for a label of this type (See sample Alchemical Analysis Program.)

Line-Continuation Symbols

BASIC numbered lines can be spread over several lines of the source code by terminating the lines with one of the line-continuation symbols: **":"** or **"\"**. This makes the lines much more readable and reduces the chance for errors.

The **":"** symbol is used to indicate the end of a BASIC statement and that there is another BASIC statement following on the next line. The "\" symbol is used to split a single BASIC statement over two or more lines.

For example, the following code uses eight BASIC line numbers:

```
A=B*C
FOR X=1 TO 120
        A=B+C*SQR(123+B)
        GOSUB 230
        FOR K=1 TO 3
                J(K)=A+K
        NEXT K
NEXT X
```

and could be rewritten as

```
A = B * C:
FOR X = 1 to 120:
        A = B + C * SQR(123 + B):
        GOSUB Sum_a_vals:                    ' Call the subroutine
        FOR K = 1 TO 3:
                J(K) = A + K:
        NEXT K:
NEXT X
```

Because all (except the last) of the above program lines end with a **":"**, the compiler continues them on the next line. Thus a single BASIC line number would be assigned saving 7 BASIC line numbers (a total of 14 bytes).

As a second example, the following line:

```
IF X > Y * Z THEN GOTO FINISHED
```

can be written as the following to improve source-code readability

```
IF X > Y * Z THEN                \
        GOTO FINISHED
```

or (note that the **THEN** is not required when followed by a **GOTO** or **GOSUB**)

```
IF X > Y * Z              \
        GOTO FINISHED
```

In these cases, we have to use the **"\"** line-continuation symbol because we cannot end the statement after the "**IF X>Y THEN**" with a colon.

If the "\" symbol appears within quotes, it is interpreted as the Yen Symbol (¥). For example, the line

```
PRINT "THIS IS A \ LINE"
```

would display or print

```
THIS IS A ¥ LINE
```

If the line were written as

```
PRINT "THIS IS A \
LINE"
```

the compiler will report the first line as an error because quotes are unbalanced and the second line as an error because it doesn't start with an BASIC verb. The line-continuation symbol **"\"** is ignored because it occurred within quotes.

Maximum Program Sizes

SBC will report an error if you attempt to compile a program that is too large for the card size specified (the compiler will identify the program line which exceeded the specified card size). It takes into account the system area used by the calculator. If the compilation is successful, the amount of memory required and the remaining free space is shown.

Maximum program sizes for the PC-1270 are:

Flash Card	8144 bytes	(program size only) (data size is 7392 bytes)
2K Card	1200 bytes	
4K Card	3248 bytes	
8K Card	7344 bytes	
16K Card	15536 bytes	

The maximum program sizes for the PC-1250 Calculator Series are:

PC-1250	1440 bytes
PC-1250A	3488 bytes
PC-1251H	9632 bytes

(All PC-1250 series programs are compiled for a 16K RAM card, programmed onto the card, and then transferred from a PC-1270 to a PC-1250 series calculator.)

Even though the program compiles successfully, it may terminate with **ERROR 6** if the combined size of the program and dimensioned arrays exceed the above limits. If this occurs, make sure you have properly dimensioned the arrays (especially the A() array). See Arrays and Flexible Memory Management.

Passwords & Copy Protection

SBC supports the Sharp S'BASIC copy protection system and automatically makes a password-protected object file unless you set the **"U"** option switch. Programs that are password-protected cannot be decompiled by SBC. In addition, password-protected programs installed in a PC-1270 or PC-1250 series calculator cannot be viewed, listed to a printer, or transferred to a cassette tape or another calculator.

Although a program can be "password protected", there is no actual password. The PC-1270 simply stores the fact that the program is "password protected" and refuses to allow the program to be viewed or copied. Once a password-protected program has been installed in the PC-1270, the "password" cannot be removed.

For more information, See Transferring Programs to Cassette Tapes and/or Other Calculators.

REM Statement and ' Comment Symbol

To force a remark to appear in the compiled code, use the BASIC keyword **REM**. The compiler will include the **REM** statement and all characters (except for trailing spaces) that follow it (up to the first comment symbol '). This is useful for including copyright notices, version numbers, and other material that you want to include in the compiled code.

For example, the line

```
REM COPYRIGHT 1994 SUPERSOFTWARE, INC.
```

will appear in the compiled code in exactly the same way

```
REM COPYRIGHT 1994 SUPERSOFTWARE, INC.
```

The compiler also allows the comment symbol **(')**. If the **(')** comment symbol occurs in a line (but not within a quoted string), it and all characters after it on the same line are discarded by the compiler. By using the **(')** symbol, you can include comments in your source file that are not included in the compiled program and therefore take no memory space in the target calculator.

Thus the line

```
PRINT A * B / C          ' total number of widgets
```

will be compiled as

```
PRINTA*B/C
```

A line that starts with the **(')** comment symbol is discarded by the compiler, e.g.,

```
' ******************************************************
' *     Alchemical Analysis Program - Version 3.0
' ******************************************************
```

will be discarded entirely.

Be sure to keep your original source code if you use the **(')** comment symbol. The comments that follow the **(')** symbol are not incorporated into the compiled program and are therefore not retrievable by decompiling the object code.

If a line contains a line number immediately followed by the **(')** comment symbol and 0 or more other characters, the line will be reported as an error. This is because the line number is unnecessary and consumes valuable space in the compiled code. The comment can be placed in the source code with just the **(')** comment symbol, taking no space in the compiled code. For example, the line

```
100 ' This is starting point
```

will be reported as an error because the line could be stated as

```
' This is starting point.
```

The elimination of the line number for the empty line from the compiled code saves three bytes (two for the line number and one to terminate the line).

Note that the **(')** symbol can only be included in your source code to indicate the beginning of a comment. If you attempted to compile the statement

```
PRINT "AB'C"
```

the **(')** would be reported as an illegal symbol and cause an error. The compiler will not treat it as a remark symbol if it is within a quoted string.

The compiler would treat everything after the **(')** in the following line as a comment:

```
PRINT "ABC" ' "ABX"
```

When BASIC lines were numbered, it was a common practice to use a numbered line with just a REM on it to show the start of a subroutine, i.e.,

```
50  REM
60  R = T + 8 : RETURN

90  GOSUB 50 : ...
```

Since the introduction of SBC Labels, this practice is no longer useful and does consume some unnecessary space. You can now write

```
R_sub:
        R = T + 8 : RETURN

        GOSUB R_sub
```

Legal Symbols

S'BASIC permits the following symbols in the source code:

" "	Space (Hex 020)
!	Exclamation point
"	Double quote
#	Number sign
%	Percent sign
&	Hex prefix (e.g., &40 = 64) and symbol
'	(Only allowed for comments - see Use of the REM Statement and Comment Symbol)
(	Left parenthesis
)	Right parenthesis
*****	Multiplication operator
+	Plus sign and addition operator
,	Comma
-	Minus sign or dash
.	Decimal point or period
/	Division operator or slash symbol
0-9	Digits 0 thru 9
:	End of statement or colon symbol
;	Semicolon
<	Less than
=	Equal sign
>	Greater than
?	Question mark
@	Commercial at sign
A-Z	Upper-case letters. Lower-case letters used in source files are converted to upper case by the compiler.
[	Square-root **(√)** symbol and function. S'BASIC also has the identical function SQR.
****	Japanese Yen **(¥)** symbol and line-continuation symbol.
]	Pi **(π)** symbol and constant. S'BASIC also has the constant PI which has the identical value.
^	Power symbol and operator (as in X^Y).
_	Underscore. Symbol that can be used only within quotes or within SBC Labels.

If any symbols appear in your source code that are not listed above, they will be reported as errors.

SBC Labels must start with a letter and can contain only letters, numbers and the underscore symbol "_".

Compiling a Program

Command Line Switch Options

The command-line format for using SBC is:

SBC [-options] (file1) [file2 [file3 [...]]]

where **"file1", "file2", ...** are the names of the source files to be compiled or the compiled file to be decompiled. **"file1"** is required. The input file extension defaults to **".LST"**. The output file will have the extension **".SBC"** (for S'BASIC Compiled). If any of the output files already exist, they will be overwritten if the compilation is successful. Options are as follows.

-nn Change default card size. The default card size is 15 representing an 8K Flash card bank. If the card size is not specified in the source file with the **#CARD_SIZE** directive, it will default to 15 unless a different value is specified on the command line with this option.

-a Produces a report of all arrays dimensioned and their space requirements.

-d Decompiles file1, file2, ... The input file extension is required. The output file will have the extension **".DCP".** Any existing **".DCP"** file will be automatically overwritten. Programs that are password protected cannot be decompiled.

-g Generates a series of file names for each program bank of a multi-bank flash card program. The command **"SBC -g xray.lst"** will compile the files **"xray_0.lst", "xray_1.lst", "xray_2.lst", ...**.

-h Same as the **"-g"** option above plus the output files are linked using BANKS.EXE into a single output file.

-i Allows implied multiplication. See Implied Multiplication.

-l Lists all program labels with the assigned BASIC line numbers.

-m Suppress non-ascending line-number warning.

-n Prevents the automatic assignment of BASIC line numbers.

-q Quiet flag. Only error messages appear on the screen. Unnecessary or warning messages are suppressed. (Can be useful in batch jobs.)

-t Saves preprocessor temporary files (**file1.pp1**, **file1.pp2**, & **file1.pp3**).

-u Make an unprotected version. Password protection is automatically applied to the compiled code unless you use this option.

-w Always reports the free memory as a warning. Default is to issue the warning when 10% or less is free.

To compile the program **MYPROG.LST**, use the following command:

SBC MYPROG

If the file **MYPROG.SBC** already exists, it will be overwritten with the new compilation. The compiled program will automatically be password-protected because the **"-U"** switch was not

specified. Unless the card size is specified in the source file, it will default to 15 for an 8K Flash card bank.

To compile the program **AJAX.BAS** without password protection, use the following command:

SBC -u AJAX.BAS

Note that we had to specify the full name of the source file because it does not have the default source file extension of **".LST"**. This copy will NOT be protected because we have specified the **"U"** switch. The output file will be named **"AJAX.SBC"**.

How SBC Compiles Programs with No Line Numbers

This version of SBC can compile source files with no BASIC line numbers. This provides the programmer with considerable freedom as he or she no longer has to be concerned with assigning line numbers or, in most cases, about running out of line numbers.

Because the BASIC language in the PC-1270 (and other Sharp calculators) requires line numbers to execute a program, it is not possible to execute a program without line numbers. In order to provide the freedom of no line numbers and meet the requirements of the BASIC language, the compiler allows the use of SBC Labels within the source file to locate the destination of **GOTO**, **GOSUB**, and **RESTORE** statements. The statements reference these labels instead of the actual line numbers.

At compile time, the compiler actually numbers each BASIC line starting with 1 and incrementing by 1. This gives you 999 program lines within each program or Flash card bank. After the file is numbered, all the label references are substituted with the proper line number.

The traditional space-saving rules all still apply -- you should use the line continuation symbols **":"** and "\" wherever possible to put as many BASIC statements as possible on each line.

Labels take no space in the compiled program file because they are replaced by conventional line numbers at compile time. Because the numbers are assigned sequentially starting with 1, they actually save space over the conventional method of numbering lines by 10's. (This is due to the space used to store the line numbers in **GOSUB** and **GOTO** statements. Because the first 99 lines used are all two digits, **GOSUB** and **GOTO** references only use two bytes for the target line number, e.g., **GOSUB 33**, thus one byte is saved for each **GOSUB** and **GOTO** in the program.

It is possible (but not easy) to run out of line numbers -- very complex PC-1270 programs rarely use more than 150-200 numbered BASIC lines. The usual problem with source files having numbered lines is that the line numbers become bunched up in areas where you need to add program code. If you numbered the lines by 10, you would have only 99 lines available. Because it numbers the file each time it is compiled, the compiler can number the lines by 1 making all 999 line numbers available. You never have to worry about space to add new program code or moving sections of your program from one place to another.

The actual line number assigned to a particular line may change each time the program is compiled (if you have added or deleted code before it), but all the label references are substituted with the correct line numbers each time the program is compiled.

Normally, the only reason you need to know the line number is usually for debugging. You can ask for a report of all the Labels used and the line numbers assigned by using the

command-line option **"-l"** (lower case **"L"**). For those particularly nasty bugs that just won't go away, you can decompile the program and obtain an exact listing of the code (including all the assigned line numbers) the PC1270 is executing.

If you inspect the Sample Source File in this Reference Manual, you can see the use of SBC Labels and the effect of the automatic numbering.

Removing the Line Numbers from a Source File

Unfortunately, there is no reliable automatic way to remove line numbers from a source file. However, to smooth the conversion to unnumbered files, the compiler is designed to allow "mixed" files where some lines have numbers and others do not.

Rather than remove line numbers from the entire file all at once (which can be a bit tedious), sections of the program can be rewritten or added without line numbers. Thus as you work on a program, it begins to lose its line numbers and eventually will have none. Of course, new programs can be written without any line numbers.

When you add an unnumbered section to a numbered program, there needs to be enough line numbers available between the numbered lines at the start and end of the new section. For example, if we want to add some code between lines 90 and 100, we could only add nine BASIC lines (which would be numbered 91 through 99). If we add more than nine lines, we will trigger the non-ascending line number warning because the assigned numbers would reach 100 or higher creating a non-ascending order of line numbers. In this case, we would have to change line 90 to a smaller number or change line 100 to a larger number.

```
90      A = B + C : R = A * 2

        (new code to be added here)

100     PRINT "THIS IS SUB 100" :
```

Liberal use of line-continuation symbols allows you to add quite a bit of code without using many line numbers. The Sample Source File in this manual has a total of 129 lines in the source file, of which 51 lines are actual code) but uses only 9 BASIC line numbers. (See also Sample Decompiled Program and Installing a Name in the PC-1270.)

If your editor has a search and replace function (which most do), you can try searching for a line number and replacing it with a label, e.g., if a subroutine starts at line 100 and there are three calls to it (**GOSUB 100**), the **"100"** could be searched for and replaced with the label **"Label_100"**. One has to be careful not to replace any **"100"**'s found that are not references to the line number 100.

You can have both a label and line number if you do not wish to remove the line numbers right away or need to specify the number, e.g.,

```
Label_100:
        100     PRINT "THIS IS SUB 100" :
                ...
```

This forces the compiler to use line 100 for **Label_100**. Subsequent lines will be numbered 101, 102, ...

The maximum gain in space savings will be obtained by removing all the line numbers wherever possible.

Sample Source Listing

```
' ************************************************************
' *
' *        Alchemical Analysis Program ALCHEM3.LST - V3.0
' *
' ************************************************************

' Program to estimate how many troy ounces of gold can be
' produced from the entered number of pounds of lead by using
' the new Version 3.0 process.  Unfortunately, the specific
' formulas for the process have been lost.

' V3.0 Eliminates BASIC line numbers and therefore produces more
'      gold because programs can be written in less time.

    #ARRAY_SIZE 182                  ' for Z$(9)*1    9 *  1 + 6 =  16
                                ' for I$(1)*80   2 * 80 + 6 = 166
                                '                            ---
    #CARD_SIZE  15                   '                Total    182

    ' Variables used
    '     E = error number (selects message number)
    '     G = ounces of gold
    '     I = luck index
    '     K = tmp
    '     L = pounds of lead

    ' We want the first user-defined function key on the PC-1270
    ' to start this program so we include the "A" S'BASIC label

Start:
    "A":                          ' Key 1 Upper Row
    clear :
    dim z$(9)*1,     \              ' First array for Flash card
       i$(1)*80 :                  ' Array for error messages

    pause "ALCHEM 3.0" :
    input "LBS OF LEAD? "; L :         '  Amount of lead to convert

    L = abs L :                      '  Strip sign, if there
    if L > 10        \
      let E = 1 :
      gosub Disp_msg :               '  Must process no more than
      goto Start                     '  10 lbs at a time.
```

```
Get_luck:
    I = 0:                          '  Clear previous luck index
    input "LUCK INDX 0-10?"; I :       '  We need some luck for
                                    '  this to work
    I = abs I :                      '  No negative numbers (too unlucky)
    if I > 10       \                '  Check limit
      let E = 2 :
      gosub Disp_msg :
      goto Get_luck

    random :                         '  Randomize the seed
    I = I / 10 * rnd(0) :            '  Modify the luck index
    G = L           \
     * sqr(8) /  8   \               '  Ver 3.0 secret formula
     * I           \
     * 12 :                         '  troy ounces/lb

    print "GOLD (OZ): "; \
    int(G * 10) / 10 :               '  Round answer to tenths
    goto Start                       '  Do it again

'
'  Scroll the message whose number is passed in E
'

Disp_msg:
    restore Disp_data

Disp_msg_read_loop:
    read I$(0) :
    K = val left$(I$(0), 2) :          ' get the message number
    if K <> E and K > 0 \              ' if not desired message and
      goto Disp_msg_read_loop          ' not the end, read again

                                    ' format with leading spaces
                                    ' & err no at end.

    I$(1) = "       " + mid$(I$(0), 3, 80) + " (" + str$ E + ")" :

    wait 24 :                        ' 24/64ths of sec to wait each print
    K = 1                            ' start at 1st char

Disp_msg_show_loop:
    i$(0) = mid$(I$(1), K, 16) :       ' get a 16-byte segment

    if len I$(0) <= 8 \                ' if segment <= 8, we're done
      wait :                         ' cancel the wait
      return

    print I$(0) :                    ' show segment
    K = K + 3 :                      ' increment 3 chars at a time
    goto Disp_msg_show_loop

'*  Error messages can be up to 67 bytes long.  First 2 bytes are the
'*  error number.  Last message must have -1 as the error number.

Disp_data:
    data "01DON,T BE GREEDY! LEAVE SOME FOR THE REST OF US!" , \
        "02COME ON! NOBODY IS THAT LUCKY!",                 \
        "-1ERROR"
```

The above source file is over 4400 bytes long, but the program compiles to an object file size of only 500 bytes. There are 129 lines in the source file but the program uses only 9 BASIC line numbers. Virtually all the documentation of a program can be kept in the source file without taking any additional space in the target card.

Decompiling a Program

Object files that are not password-protected can be decompiled to produce an ASCII text file by using the **"-d"** switch option (See Command Line Switches). The primary purpose of this feature is to allow you to convert existing programs to a text file for editing.

When an object file is decompiled, SBC produces a text file with the same filename as the object file but with the extension **".DCP"**.

If you decompile an object file that was compiled by SBC, none of the "white-space" or remarks (except for remarks following REM statements) in the original source file will appear. In addition, none of the formatting you have incorporated into the source file will appear. The decompiler will insert continued-line symbols "\" if a line exceeds approximately 65 characters, but they will not appear in the same place as in the original source code. For these reasons, it is important that you keep your original source file.

If you decompile programs compiled with other products and SBC finds unknown binary codes, it will place a "¿" in the decompiled source code.

Sample Decompiled Program

If we compile the above program and then decompile it, we get the following listing:

```
1 "A" : CLEAR : DIM Z$(9)*1,I$(1)*80 :
    PAUSE "ALCHEM 3.0" : INPUT "LBS OF LEAD? ";L :
    L= ABS L : IF L>10 LET E=1 : GOSUB 4 :
    GOTO 1
2 I=0 : INPUT "LUCK INDX 0-10?";I : I= ABS I :
    IF I>10 LET E=2 : GOSUB 4 : GOTO 2
3 RANDOM : I=I/10* RND (0) : G=L* SQR (8)/8*I*12 :
    PRINT "GOLD (OZ): "; INT (G*10)/10 :
    GOTO 1
4 RESTORE 9
5 READ I$(0) : K= VAL LEFT$ (I$(0),2) :
    IF K<>E AND K>0 GOTO 5
6 I$(1)="       "+ MID$ (I$(0),3,80)+" ("+ STR$ E+")" :
    WAIT 24 : K=1
7 I$(0)= MID$ (I$(1),K,16) : IF LEN I$(0)<=8 WAIT :
    RETURN
8 PRINT I$(0) : K=K+3 : GOTO 7
9 DATA "01DON,T BE GREEDY! LEAVE SOME FOR THE REST OF US!", \
      "02COME ON! NOBODY IS THAT LUCKY!", \
      "-1ERROR"
```

S'BASIC Statements & Functions

In the following notations, (expr) is a real number or an expression that will resolve to a real number, (string) is a character string or an expression that will resolve to a character string, (iexpr) is an integer or an expression that will resolve to an integer, and (var) is a real or string variable.

Arguments, which can be constants, variables, or expressions, in parentheses "(..)" are required (although the parentheses themselves are not required unless the argument is an expression). Arguments in square brackets "[..]" are optional and may be omitted. Only one of a list of arguments separated by "|"'s is used.

ABS (expr) Returns the absolute value of expr.

ACS (expr) Returns the arccosine of expr. Inverse function is **COS**. See also **DEGREE**, **GRAD** and **RADIAN**.

(..) AND (..) Binary and logical AND. If arguments are logical expressions enclosed in parentheses, a logical AND is performed and the result is 1 if true, or 0 if false, e.g.,

(5>1) AND (9=4+5) returns 1 (true).

If the arguments are numeric only, a binary AND is performed, e.g.,

15 AND 8 returns 8

Mixed expressions are solved by evaluating the logical expression first and then performing a binary AND with the numeric argument, e.g.,

2 AND (1=1) returns 0

AREAD (var) Reads the contents of the display into var when program is started with a [DEF] key.

ASC (string) Returns the ASCII code of the first character in string. Inverse function is **CHR$**.

ASN (expr) Returns the arcsine of expr. Inverse function is **SIN**. See also **DEGREE**, **GRAD** and **RADIAN**.

ATN (expr) Returns the arctangent of expr. Inverse function is **TAN**. See also **DEGREE**, **GRAD**, and **RADIAN**.

CHR$ (iexpr) Returns the character that has an ASCII value of iexpr which must be from 32 to 95. Inverse function is **ASC**.

CLEAR Removes all DIMensioned arrays from memory and resets the fixed variables A-Z to 0's (or A$ to Z$ to nulls). In multi-bank programs, **CLEAR** should only appear in the first bank (Bank 0).

COS (expr) Returns the cosine of angle expr. Inverse function is **ACS**. See also **DEGREE**, **GRAD**, and **RADIAN**.

CSAVE [[fname], password]

Sends the program in the calculator to a cassette tape or another calculator (using the EA-129C cable). The password is up to a 7-byte string in quotes, e.g., **"PASS123"**. Both the filename and password are optional. If the filename is omitted but a password used, the separating comma is required, e.g., **CSAVE, "PASS123"**. If the program in the originating calculator is password-protected, the **CSAVE** command is ignored. See also Passwords & Copy Protection and Transferring Programs to Cassette Tapes and/or Other Calculators.

DATA data, data, ...

Identifies the following information to be read by the **READ** command. See also **READ** and **RESTORE**.

Data can be numeric or string constants, or SBC Labels, e.g.,

```
DATA PROC_1, 12, "ABC"

READ B, C, D$ :        ' Get sub label
GOSUB B :              ' Execute subroutine
...
```

DEG (expr)

Converts expr from degrees, minutes & seconds to decimal degrees (dd.mmsss to dd.dddddd). Inverse function is **DMS**.

DEGREE

Trig functions expect & return angles in degrees. See also **GRAD** and **RADIAN**.

DIM X(size1[, size2]) or Y$(size1[, size2]) [*len]

Establishes an array of real numbers or strings. Size1 is the number of rows, not to exceed 255, size2 is the number of columns, not to exceed 255. Rows and columns start with 0, e.g., the first element in the first row in the array B(2, 2) is B(0, 0).

For arrays of strings, the length of the string can be specified. For example, **DIM X$(23)*64** dimensions an array of 24 strings (the first being X$(0)) each 64 bytes long. The maximum length of a string is 80 bytes, the default length is 16 bytes.

All values of an array are reset to 0's or null's when it is dimensioned. If you attempt to dimension an array that is too large for the calculator's memory, **ERROR 6** will occur.

The **DIM** statement can be used for the special A array. However, the A array is always a single-dimension array and the first row is always 1. There are other special rules concerning the A array.

For multi-bank Flash card programs, the **DIM** statement should appear only in the first bank (Bank 0).

See also Arrays and Flexible Memory Management.

DMS (expr)

Converts expr from decimal degrees to degrees, minutes & seconds (dd.dddddd to dd.mmss...). Inverse function is **DEG**.

(..) EEX (..)

Represents the **"E"** used in scientific notation numbers, e.g., **1.234 EEX 12** represents the constant 1.234 E 12.

The first number is optional and defaults to 1 if omitted, e.g., to divide a number by 1000, the following is legal:

```
(number) / EEX 3
```

END

Stops execution at the end of a program. More than one **END** can appear in a program. **END** is not required as the last of the program. See also **STOP**.

EXP (expr)

Returns the constant e raised to expr power, i.e.,

EXP(1) returns 2.718281828

FOR (expr1) = (expr2) TO (expr3) [STEP expr4]

Executes all statements between the **FOR** and the following **NEXT** as long as **expr1** is less than or equal to **expr3**. Increments **expr1** from an initial value of **expr2** to a value of **expr3** by **expr4**. Default **expr4** is 1. See also **NEXT** and **STEP**. The value of **expr** must range from -32768 to 32767. (**STEP** cannot be 0 but can be positive or negative.)

GOSUB (label | lineno)

Starts executing the program at label or lineno. Returns to the statement following the **GOSUB** when a **RETURN** is encountered. “label” can be an SBC Label or S’BASIC Label, "lineno" is a BASIC line number. E.g.,

```
GOSUB Print_total: ...

Print_total:
  PRINT "....":
  RETURN
```

GOTO (label | lineno)

Starts executing the program at label or lineno. See also **GOSUB**.

GRAD

Trig functions expect & return angles in grads. See also **DEGREE** and **RADIAN**.

IF (condition) [THEN] (verb) (statements)

Performs (statements) if (condition) is true, e.g.,

```
IF 1 < 2 PRINT "1 IS LESS THAN 2"
```

would print "1 IS LESS THAN 2". (verb) is a command, e.g, **PRINT, GOTO, LET, ...** . The keyword **THEN** is usually not required, e.g.,

```
IF A>12 THEN LET A=12
```

can be written

```
IF A>12 LET A=12.
```

If the (condition) is false, all statements are skipped until the start of the next BASIC line number.

There is no **IF..THEN..ELSE** construction, but you can effectively create one by using the following technique. State the ELSE condition first and then use the IF..THEN construction. For example, if we wanted to write

```
IF A > B      \
  LET X = A   \
ELSE          \
  LET X = B
```

we can use the following instructions:

```
X = B :
IF A > B      \
  LET X = A
```

INKEY$

Returns a 1-byte string representing the last key pressed. Returns a null if no key (or ENTER) was pressed, therefore the value must be tested in a loop, e.g.,

```
Inkey_loop:
  A$ = INKEY$:         ' Get character
  IF ASC A$ = 0  \     ' If nothing there
    GOTO Inkey_loop    ' read it again
```

INPUT [(string) (; l ,)] (var)

Suspends execution of the program and accepts a number or string terminated with [ENTER] and places its value into var. The user can be prompted with the optional (string). If the prompt string is terminated with a **";"**, the prompt remains on the screen and scrolls to the left as the entry is made. If terminated with a **","**, the prompt string disappears as soon as the first character is entered.

If no entry is made (other than pushing the [ENTER] bar) all statements following the INPUT statement are skipped until the next BASIC line number is encountered.

If the [YES] key is pushed, the character "Y" is assigned to the variable. Similarly, if [NO] is pushed, the character "N" is assigned to the variable.

On the PC-1270 only, a string variable can be included in the prompt string by including the variable name in semicolons within the prompt string. This must be a variable and not a constant. For example, the following line will display the string variable B$ in the prompt string:

```
B$ = "SHARP" :          ' Default value
INPUT "BEST CALC (;B$;)? "; B$
```

The prompt will appear as:

```
BEST CALC (SHARP)?_
```

To prompt a numeric variable, you can either convert it to a string first, e.g.,

```
N = 48 :                ' Default term
A$ = STR$ N:
INPUT "TERM (;A$;)? "; N
```

which will appear on the screen as

```
TERM (48)?_
```

You can also use the USING format command. The variable is formatted according to the most recent **USING** statement executed (if none, then the default formatting applies). See **USING**.

INT (expr) Returns the integer value of expr.

LEFT$ (string, iexpr) Returns the iexpr left bytes of string. See also **RIGHT$**.

LEN (string) Returns the length of string in bytes.

LET Assigns a value to a variable, e.g., **LET A = 7**. **LET** is not required except in the **IF** construction, e.g., **IF X > 2 LET Y = 7**.

LN (expr) Returns the natural logarithm (to the base e) of expr. See also **LOG**.

LOG (expr) Returns the logarithm to the base 10 of expr. See also **LN**.

LPRINT [USING [string];] (var | string[, | ; var | string])

Prints the following list of variables and/or strings on the attached printer. Output is formatted according to the most recent **USING** statement executed (if none, then the default formatting applies). See **USING**.

MEM Returns the free memory in the calculator in bytes.

MID$ (string, iexpr1, iexpr2)

Returns a substring of string **iexpr2** bytes long starting at byte **iexpr1**, e.g,

```
MID$("ABCDE", 3, 2)
```

returns "CD".

NEXT (expr) Used to terminate a **FOR** loop. When **NEXT** is encountered, the program will return to the corresponding **FOR** statement.

NOT (..) Binary NOT. **NOT 0** equals -1, and **NOT -1** equals 0. Flips all bits in argument. Cannot be used in the logical sense, i.e., although the expression **(2=2)** evaluates to 1, **NOT (2=2)** evaluates to -2.

ON (iexpr) (GOTO | GOSUB) (label | lineno, label | lineno, ...)

If **iexpr** equals 1, control is passed to the first label or lineno, if equal to 2, the second label or lineno, and so forth. If **GOSUB** is used, control returns to the next statement when a **RETURN** is encountered.

```
ON X GOSUB TYPE_1, TYPE_2, TYPE_3
```

will branch to the label **TYPE_1** if X has a value of 1, to **TYPE_2** if x has a value of 2, or to **TYPE_3** if X has a value of 3. If X has a value greater than 3 or less than 0, the program will execute the next BASIC statement.

(..) OR (..) Binary and logical OR. See **AND** for more information.

PASS (string) Used to protect a program in a PC-1250 series calculator with the password (string). If the program is already protected, issuing this command with the same password that was used to protect the program will remove the protection. (string) can be up to 7 characters. Valid only as a command in the PC-1250 Series (i.e., cannot be used within a program).

PAUSE [USING [string];] (var | string [(,|;) var | string])

Displays the variable or string for about 0.85 seconds. If two arguments are separated by a comma, the screen will be separated into two halves (8 bytes each on a PC-1270, 12 bytes each on a PC-1250A or attached printer). Output is formatted according to the most recent **USING** statement executed (if none, then the default formatting applies). See **USING**.

PI Returns the constant value π (3.141592654).

PRINT [USING [string];] (var | string) [(,|;) var | string])

Displays variables or strings unless the statement **PRINT = LPRINT** has been executed, in which case, the items are printed on an attached printer. Output is formatted according to the most recent **USING** statement executed (if none, then the default formatting applies). See **USING**.

PRINT = LPRINT — A special form of the **PRINT** statement that sends all **PRINT** statement output to a printer, if attached, or to the display if no printer is attached. (Use the statement **PRINT = PRINT** to cancel the **PRINT = LPRINT** statement.)

PRINT # [fname l varlist] l [fname; varlist]

Outputs data to a cassette tape.

RADIAN — Trig functions expect & return angles in radians. See also **GRAD** and **DEGREE**.

RANDOM — Changes the random number seed used by **RND**. See also **RND**.

READ var [, var ...] — Reads data items from **DATA** statements and assigns it to the variables specified. Items can be separated by commas to read a list of data, e.g., **READ A, B, C** would assign the first value in the **DATA** statement to A, the second to B, and the third to C. The data type must match the variable type.

Reading starts at the first **DATA** statement in the program unless the **RESTORE** command has been used to set the data pointer. The data pointer increments by one data item for each variable read.

REM [string] — Places the string as a remark in the compiled object code. See Use of the REM Statement and ' Comment Symbol.

RESTORE [label l lineno]

Resets the READ pointer to the label or lineno specified. If none specified, to the first DATA statement in the program.

RETURN — Used at the end of a subroutine to return control to the statement following the invoking **GOSUB**.

RIGHT$ (string, iexpr) — Returns the rightmost iexpr bytes of string. See also **LEFT$**.

RND (expr) — Returns a random number. If **expr** is greater than or equal to 0 and less than 1, a decimal random number will be returned that is greater than or equal to 0 and less than 1.

If **expr** is greater than 1, the random number will be an integer greater to or equal to 1 and less than or equal to **expr**.

See also **RANDOM**.

SGN (expr) — Returns 1 if **expr** is positive, -1 if **expr** is negative, or 0 if **expr** is 0.

SIN (expr) — Returns the sine of angle **expr**. Inverse function is **ASN**. See also **DEGREE**, **GRAD**, and **RADIAN**.

SQR (expr) Returns the square root of **expr**, which must be positive.

STEP Used in a **FOR...NEXT** loop to determine the step interval if the interval is other than +1. For example,

```
FOR K = 120 TO 10 STEP -10
  .
  .
NEXT K
```

decrements the value of K by 10 for each pass through the **FOR...NEXT** loop until K is less than 10.

STOP Halts the execution of a program for diagnostic purposes. **STOP** in program line 10 would cause **BREAK IN 10** to appear in the display. See also **END**.

STR$ (expr) Returns a string equivalent to **expr**, i.e, **STR$ 1.23** returns "1.23". See also **VAL**.

TAN (expr) Returns the tangent of angle **expr**. Inverse function is **ATN**. See also **DEGREE**, **GRAD**, and **RADIAN**.

THEN Used in the **IF (condition) THEN (statement)** construction. The word **THEN** is generally not required but can be used for clarity.

TO As in **FOR X = 1 TO 12**. See **FOR** and **NEXT**.

USING [string]

When used with (string), formats the output for the next **PRINT**, **LPRINT**, **PAUSE**, or **INPUT** statement. When used alone without (string), clears any existing **USING** format statement causing subsequent **PRINT** statements to use default formatting.

(string) can be a literal string or a string variable and is a formatting string containing the following characters: **"#[#]"** represents a right-justified numeric field; **"."** represents the decimal point; **"^"** represents scientific notation for printing numbers; and **"&[&]"** represents a left-justified string field.

```
A = 125:
X$ = "ABCDEF":
USING "##.##^" :      ' Scientific notation
PRINT A
```

Displays 1.25E 02

```
USING              ' Use default
PRINT A
```

Displays 125

```
USING "&&&&&&&&" :
PRINT X$
```

Displays ABCDEF

```
F$ = "####&&&" :
PRINT USING F$; A; X$
```

Displays 125ABC

VAL (string)

Returns the equivalent numeric value of string. Converts the leading numeric characters of string stopping at the first non-numeric character, i.e., **VAL "123JLL56"** returns 123.

WAIT [iexpr]

Normally the calculator will stop executing each time a **PRINT** statement is executed until [ENTER] is pushed. The **WAIT** statement with an argument will cause the calculator to continue executing the program after waiting an interval of time determined by the value of **iexpr**. **iexpr** can range from 0 to 65535. A value of 64 will pause about 1 second, a value of 3840 will pause for about 1 minute. The value cannot exceed 65535 (which would cause a delay of about 17 minutes).

WAIT with no argument resets the calculator to its normal condition of waiting until **[ENTER]** is pushed.

Error & Warning Messages

Fatal Compiler Errors

The following errors are immediately fatal and are reported to the screen.

Error nn: Insufficient memory.

SBC can't get enough memory to run. If you are using more than one program simultaneously or TSR (terminate and stay resident) programs, try SBC alone to see if the error continues. SBC requires 256K or less of memory to run.

Error 3: Can't find input file (fspec).

SBC cannot find the input file specified in the command line.

Error 4: Input and output files have same name.

The input and output files have the same name. This can occur if you use the output file extension type for the input file, e.g., the command **SBC MYFILE.SBC** attempts to instruct SBC to compile the file **MYFILE.SBC** and put the compiled code into a file of the same name.

Error 8: Illegal default card size (n) specified.

The card size specified on the command line is not a legal size.

Error 9: Can't find Bank 0 file 'file' and -g or -h option specified.

The **"-g"** or **"-h"** option is specified and the compiler cannot find the bank 0 file indicated.

Error 12: Can't close output file (fspec).

SBC can't close the output file.

Error 13: Can't open input file (fspec).

SBC can't open the input file. The file may be locked by another process.

Error 15: Can't open output file (fspec).

SBC can't overwrite the existing output file. It may have a read-only attribute or be locked by another process.

Error 19: Input file (fspec) incomplete.

Occurs when attempting to decompile a file that is not a complete program.

Error 21: Input file (fspec) corrupt.

Occurs when attempting to decompile a file that is either a corrupted program or not an S'BASIC compiled file.

Error 22: Can't decompile protected program (fspec).

The file that you are attempting to decompile is password protected. You must make an unprotected version of it if you wish to decompile it. See Command Line Switch Options.

Error 23: Source file name can't exceed 5 chars to generate file names.

This can occur when the "-g" option is used which automatically generates a series of filenames (one for each bank in a multi-bank

program) to be compiled. Because the compiler automatically addes the suffixes “_0”, “_1”, ... up to “_63”, the root file name cannot exceed 5 characters.

Error : Over xx BASIC lines in Demo version.

The demonstration version of the program will not compile source files that have more BASIC lines than the number indicated.

Error 20: (fspec) compilation aborted.

SBC has found errors when attempting to compile the file (fspec) and did not produce a compiled output file.

Non-fatal Compiler Errors

Non-fatal errors do not stop the processing of the source file being compiled but will prevent an object file from being produced. These errors are reported to the screen as shown in the following example.

Attempting to compile the following line from a source file named **MYPROG.LST**

PRINT "TOTAL: "; (A + (C * D)

will cause the following errors to be reported

Compiling 'MYPROG.LST'...
MYPROG.LST(12) : Error: Expected a ')' in line 5.
MYPROG.LST(1) : Error: ‘MYPROG.LST’ compilation aborted. 1 error exists.

The **"12"** in "MYPROG.LST(12)" is the actual line number of the source file in which the error occurred. The **"line 5"** at the end is the BASIC program line number (if no numbers are in the source file, then this is the number assigned by the compiler during the compilation). The second error is a general error that indicates no output file was produced.

If you are using the MS-DOS EDIT program to edit your program, the source-file line number will appear in the lower right-hand corner of the screen. (Most other "text" or "programmer's" editors also will show the source-file line number.) Using this feature, you can quickly locate the line of the source file that contains the reported error.

Error: A() array used but not DIMensioned.

The compiler has detected that the A() array is used in the program but has not been DIMensioned. This occurs as an error for FLASH cards and as a warning for RAM cards.

Error: Array ‘x’ dimensioned more than once.

The indicated array has been dimensioned more than once in the program being compiled.

Error: Array ‘x’ has an invalid number of rows (or columns).

The number of rows (or columns) must be between 0 and 255 (representing 1 to 256).

Error: Array ‘x’ element length invalid (1-80 bytes).

The length of each element of a string array is less than 1 or greater than 80.

Error: Array ‘x’ size exceeds 16K.

The combination of rows, columns, and element length specified would exceed 16K.

Error: Can't continue REM in line xx.

You can't use the line-continuation symbol **'\'** to continue a REM statement.

Error: Can’t dimension both the A() and A$() arrays.

Both the A() and A$() arrays appear in one or more dimension statements.

Error: Can’t process this DIM statement.

There is an error in the DIM statement being processed and the compiler cannot process it.

Error: Can’t dimension the A() (or A$) array with less than 27 rows.

You have attempted to dimension the A() array with less than 27 rows. The first 26 rows of the A() array correspond to the built-in variables A-Z and cannot be dimensioned.

Error: Can't understand (string) in line xx.

The compiler can't understand the sequence of characters shown in the indicated line. Check the spelling of the keyword or label. BASIC statements must start with a keyword or variable assignment.

SBC Labels must start at the left margin and be followed immediately by a colon. If a Label does not start at the left margin, this error message will occur.

(If you wish to use implied multiplication, you must set the **“-i”** implied flag in the command line or you will get this error for every instance of implied multiplication. See Command Line Switch Options and Implied Multiplication for more information.)

Error: #CARD_SIZE specification ( ) is illegal.

The **#CARD_SIZE** specification in the source file is an illegal value. Legal values are 2, 3, 4, 5, 8, 9, 15 or 16. Default size is 15 for an 8K Flash card bank.

Error: #directive found after program start.

The **#CARD_SIZE** and **#ARRAY_SIZE** (if used) must appear before the first BASIC program line.

Error: Duplicate #ARRAY_SIZE directive.

More than one **#ARRAY_SIZE** compiler directive has been found in the source file.

Error: Duplicate #CARD_SIZE directive.

More than one **#CARD_SIZE** compiler directive has been found in the source file.

Error: Expected 1 to 4 hex digit(s) after '&'.

The character following the '&' is not a legal hex digit (0-9, A-F). Hex constants must be preceeded with a '&' and be within the range of **&0** to **&FFFF**.

Error: Expected n ')'s in line xx.

There are n missing right parentheses.

Error: Expected n '('s in line xx.

There are n missing left parentheses.

Error: Expected a ')' in line xx.

There is a missing right parenthesis.

Error: Expected a '(' in line xx.

There is a missing left parenthesis.

Error: Expected a verb instead of ‘ ‘ in line xx.

The compiler expected a BASIC verb instead of the word displayed. Each BASIC statement must start with a verb or a variable in an assignement statement.

Error: Expected line xx to be continued.

The last line in the source file ended with a line-continuation character (“****” or "**:**") and a new numbered line follows.

Error: First array cannot be the A() or A$() array for a FLASH card.

The first array dimensioned must be a string array and cannot use the letter A. See P*ROM Application Note Programming CF Series Flash Cards.

Error: First array element length incorrect for a FLASH card.

The length of each element in the first array is not correct for a FLASH card. See P*ROM Application Note Programming CF Series Flash Cards.

Error: First array has incorrect number of rows for a FLASH card.

The first array dimensioned must meet the specifications for a Flash card. See P*ROM Application Note Programming CF Series Flash Cards.

Error: First array must be a string array for a FLASH card.

The first array dimensioned must be a string array for a Flash card bank.

Error: First array must have only 1 column for a FLASH card.

The first array dimensioned has more than one column.

Error: FLASH card data area exceeds 8K.

The data area exceeds the maximum of 8K for a Flash card program.

Error: Illegal (') in quotes in line xx.

The comment symbol was found within a quoted string. Comments cannot be started within a quoted string.

Error: Illegal (_) in line xx.

The underscore symbol can only appear within a quoted string, e.g. the following is okay:

```
print "Loan_amount"
```

Error: Illegal char ( ) in line xx.

The illegal symbol shown was found in line xx. See S'BASIC Symbols for a list of legal symbols.

Error: Label already exists.

A label can only be declared once within a source file. This error occurs when the same word is declared as a label in more than one place in a source file, e.g., the following will cause this error:

```
Label_1:
        PRINT "THIS IS LABEL_1"

Label_1:                ' Label can't be declared twice
        PRINT "THIS IS A DUPLICATE LABEL"
```

Error: Label already exists for this BASIC line.

A label already exists for this line. You cannot have more than one label for a BASIC line, e.g., the following will cause this error:

```
Label_1:
Label_2:
        PRINT "TOO MANY LABELS"
```

Error: Label is unreferenced.

The label on this line is not directly addressed by any statements within the source file. Unless this label is addressed indirectly, you probably don't need it and can remove it.

Error: Line xx contains only a comment.

Line xx has only a comment. This line is probably unnecessary. You can remove the line and leave only the comment.
See Use of the REM Statement and ' Comment Symbol.

Error: Line xx empty.

Line xx has nothing but a line number and is probably unneeded. You can either eliminate the line number entirely, or, if the line number is required, place the **REM** statement after the line number, e.g., **120 REM**. See Use of the REM Statement and Comment Symbol.

Error: Line number 0 or missing. (xx)

Occurs if the “-n” (suppress automatic assignment of line numbers) option is used and the indicated line has no line number or a line number of 0. This error also occurs if you forget to put a line-continuation symbol at the end of the previous line when continuing a line.

Error: Line number xxxx too large (> nnn).

Line xxxx starts with a number that is too large. The maximum BASIC line number is shown as nnn.

Error: No program lines in source.

The source file did not have any BASIC program lines in it. There must be at least one numbered program line.

Error: Program too big for an 8K FLASH bank.
Error: Program too big for a nK RAM card.

The program length exceeds the specified card size. The size was exceeded during compilation at the source line number reported with the this error message. If present, the **#CARD_SIZE** specification in the source file is used for the program length test. If no **#CARD_SIZE** is specified in the source file, the default **#CARD_SIZE** specified in the command line is used. The size used is shown in the error message. If neither is specified, the default card size of 15 (for an 8K Flash card bank) is used.

Error: Program too big for a (type) calculator.

The program you are compiling is too large to fit into the PC-1250 series calculator you have specified (type is one of the following: PC-1250 (2K), PC-1250A (4K), or PC-1251H (10K)). Make sure the **#CARD_SIZE** specification in the file is correct for the calculator -- see Compiler Directives.

Error: Quotes unbalanced in line xx.

Line xx ends in an open quote. Lines cannot be continued within quotes - See Use of the Line-Continuation Symbols. However, you can concatenate strings and place the line-continuation symbol between the strings, e.g. the following is legal,

```
PRINT "ABC" +   \
    "DEF" +   \
    "XYZ"
```

If you wish to print or display the quote symbol, you must use the **CHR$** command, e.g., the following line

```
PRINT "THIS IS A QUOTE " + CHR$(&22) + "."
```

will print

THIS IS A QUOTE "

Error: Source line too long (> nnn bytes). (xx)

The source code line is too long. The maximum length of each line of source code is shown. Note, however, that BASIC lines can be virtually any length by using the line-continuation symbols. See Use of the Line-Continuation Symbols.

In practice, source code lines should be limited to 80 characters in order to fit on a screen.

Error: Unknown compiler directive.

The compiler directive on the indicated line is unknown. See **#ARRAY_SIZE** and **#CARD_SIZE**.

Compiler Warning Messages

Unless the **"-q"** (quiet) switch is set on the command line, the compiler may issue any of the following warnings. A compiled output file is created even if warnings occur.

Warning: nnn bytes free in 8K FLASH bank.

Occurs when **#CARD_SIZE** is 15 and the compiled program exceeds approximately 90% of the memory available. The number of bytes remaining is shown. If the **"-w"** option is set on the command line, this warning always appears.

Warning: nnn bytes free in 8K data area.

Occurs when **#CARD_SIZE** is 15 and the size of the arrays exceeds approximately 90% of the memory available. The number of bytes remaining is shown. If the **"-w"** option is set on the command line, this warning always appears.

Warning: nnn bytes free in xK RAM card.

Occurs when **#CARD_SIZE** is 2, 4, 8 or 16 and the compiled program plus the arrays exceed approximately 90% of the memory available. The number of bytes remaining is shown. If the **"-w"** option is set on the command line, this warning always appears.

Warning: nnn bytes free in (type) Calculator.

Occurs when **#CARD_SIZE** is 3 (for PC-1250), 5 (for PC-1250A), or 9 (for PC-1251H) and the compiled program size exceeds approximately 90% of the memory available in the specified calculator. The number of bytes remaining is shown. If the **"-w"** option is set on the command line, this warning always appears.

Warning: nnn bytes free in xK card.

Occurs when the size of the compiled program exceeds approximately 90% of the memory available in the card specified. The number of bytes remaining is shown. If the **"-w"** option is set on the command line, this warning always appears.

Warning: A() array used but not DIMensioned.

The compiler has detected that the A() array is used in the program but has not been DIMensioned. This occurs as an error for FLASH cards and as a warning for RAM cards.

Warning: Computed array size (x) differs from stated #array_size.

The compiler has computed a different array size than the one found in the #array_size directive. The specified size is used. (You should make sure the specified size is correct, change it to agree with the computed value, or eliminate the #array_size directive entirely).

Warning: Keyword 'x' not valid in PC-1270.

The keyword shown is valid only in PC-1250 series calculators. It is used in the indicated source line of the current program and the **#CARD_SIZE** specification indicates that the program is being compiled for a PC-1270.

Warning: Non-ascending line number 'xx'.

The indicated line number is equal to or less than the preceeding line number. Occurs only in numbered source files or in "mixed" source files when some of the lines are numbered.

If you are inserting an unnumbered section of code into a numbered file, you must make sure there are enough line numbers available for the new section. (The numbers are assigned by 1's, so to insert 10 lines of code, you only need 10 line numbers available.)

Warning: Unprotected program.

The PC-1270 program compiled is not password-protected because the **"-u"** switch is present in the command line. See Command Line Switch Options.

Warning: Unprotected program for (type).

This warning is to remind you that the program compiled is for a PC-1250 series calculator. The program needs to be transferred to a 16K RAM card and then transferred to the PC-1250 series calculator.

PC-1270 Run-Time Errors

These appear as **ERROR # (IN ###)** in the PC-1270 display and are system errors whose significance can be determined from the following table. The BASIC line number (IN ###) reported may not be the exact line number in which the error occurred. If there is no error in the reported line, the line with the error will be within a few lines after the reported number.

For source files with no line numbers, you can either generate a Label cross-reference table with the command-line option **"-L"** or decompile the program to inspect the actual version of the code being executed.

ERROR 1 (IN ###)	This is a syntax error. The calculator cannot solve the expression or a function call has missing arguments or arguments of the wrong type.
ERROR 2 (IN ###)	This is an underflow or overflow calculation error. It occurs because of an attempt to divide by 0 or when a number becomes too large.
ERROR 3 (IN ###)	Caused by attempting to use an array that has not been DIMensioned, to address an element that is outside of the DIMensioned range, or if you attempt to DIMension an array that already exists. Also occurs if the values in a **FOR...NEXT** loop are out of range (-32768 to 32767) or the value in a **WAIT** statement exceeds 65535.
ERROR 4 (IN ###)	Literally means the line number is too large. Occurs when the calculator cannot find the line number or label addressed by a **GOTO** or **GOSUB** statement. Also occurs when there is no program in a RAM card (no line number can be found).
ERROR 5 (IN ###)	Occurs when the nesting level is exceeded for **FOR...NEXT** loops or **GOSUB** statements. Also occurs if a **RETURN** statement is encountered without having executed a **GOSUB**, a **NEXT** is encountered without the matching **FOR** statement, or a **READ** is attempted with no or an insufficient amount of data in the **DATA** statement(s).
ERROR 6	Out of memory error. Occurs when array(s) are dimensioned when the combined length of the program plus the memory required by the array(s) would exceed the memory available in the calculator.
ERROR 7 (IN ###)	Occurs with the **USING** statement if you attempt to print or display a number with a **USING** format specification that is too small to contain it, e.g., **USING "#": PRINT 12**. (If the **USING** statement has a syntax error, **ERROR 1** will be reported.)
ERROR 8 (IN ###)	Usually occurs when using a printer with weak batteries.
ERROR 9 (IN ###)	A general catch-all error number for other errors that don't fall into the above categories. For example, the statement **CHR$(1)** will cause **ERROR 9** because it is an illegal character. If you attempt to read an element of the A array with the wrong type variable, **ERROR 9** occurs, e.g., the statements **A$ = "MAXIMUM": PRINT A** will cause **ERROR 9**.

Utilities

BANKS.EXE Linker for Multi-Bank Flash Card Programs

For multi-bank Flash card programs, the individual source files need to be linked into a single object file. This single file is an exact copy of the binary data which is then programmed onto the Flash card. Programs for RAM cards and programs for Flash cards that have only one 8K bank do not need to be linked.

```
BANKS.EXE - Link compiled SBC programs into a single file. Ver 1.xx
(C) 1994 P*ROM Software, Inc. - All Rights Reserved.
Licensed to:

Usage:  banks -options (root_name[.ext])

   (root_name) is the file name without the bank designator,
   e.g., 'spud' would read the files 'spud_0.sbc',
   spud_1.sbc, ..., spud_63.sbc.  File numbers must be contiguous.

   Default file extension is '.sbc'

   Output file is (root_name).(ext), e.g., spud.sbc

   -q   Quiet mode. No unnecessary messages on screen.
   -s   Allow non-contiguous bank numbering.
   -t   Build output file at directory pointed to by environment
        variable 'tmp'.  (default is same directory as
        input files).
   -v   Automatically overwrite output file if it exists.
```

(root_name) is the root name of the files to be linked. The extension is optional and if not specified defaults to **".sbc"**. A full path can be specified in the (root_name).

The root portion of the file name cannot exceed five characters (this is because the linker automatically generates up to 64 file names by adding the suffixes **"_0"**, **"_1"**, **"_2"**, **...,** **"_63"** and the total length of the DOS filename cannot exceed 8 characters).

For example, to link the files for the **"SPUD"** project, which are in the **\src** directory of the **C:** drive, we would issue the following command:

```
BANKS C:\SRC\SPUD
```

This will link the series of files **"SPUD_0.LST"**, **"SPUD_1.LST"**, **"SPUD_2.LST"**, ... **"SPUD_63.LST"** into the single output file **"SPUD.SBC"**.

Options:

-s Allows non-contiguous bank numbering. If banks are skipped in a Flash card, you need to specify this option to tell the linker otherwise the linker assumes the file is missing and will report an error.

-t Normally the output file is built at the same location as the input files. If this option is specified, the linker looks for an environment variable **"TMP"** and if found, will build the output file at the path specified in **"TMP"**.

BANKS.EXE Errors

If an error occurs, BANKS.EXE returns a non-zero value to DOS or the calling process, otherwise a 0 is returned.

Error: Input and output file have same name.
The input file and output file have the same name.

Error: Unable to find bank 0 file.
In a multi-bank Flash card program, there must be a file for bank 0 and none was found by the linker. This is the file with the **"_0"** extension, e.g., **"SPUD_0.SBC"**

Error: Unable to find bank x file.
BANKS.EXE expects there to be a continuous series of object files to link and there is one or more missing files. If you wish to skip bank numbers, you need to specify the **"-s"** option on the command line when you run BANKS.EXE

For example, if the files **"SPUD_0.SBC"**, **"SPUD_1.SBC"**, and **"SPUD_3.SBC"** exist (but there is no file **"SPUD_2.SBC"** because bank 2 is not used), BANKS.EXE will report this error unless the **"-s"** option is specified.

Error: Unable to open output file 'file'.
BANKS.EXE cannot create the output file shown.

Error: Unable to open input file 'file'.
BANKS.EXE cannot open the specified input file.

Error: Multi-bank program and '#array_size' not specified in bank_0.
BANKS.EXE has detected that the array size has not been specified in a multi-bank program. The program will not run properly if the array size is not set.

This error should not occur with SBC as it automatically sets the array size to the proper value.

Error: Root name 'name' too long (over 5 chars).
The root file name is greater than five characters. Because the linker automatically adds the suffixes **"_0"**, **"_1"**, **...** up to **"_63"** to the input file name to generate all the file names, the root file name cannot exceed 5 characters.

COPYLST.EXE

This utility makes a copy of all the files that have the specified root filename. Flash card projects have one source file for each bank, and you can have 4, 8, 16, or even up to 64 source files for a particular project.

The naming conversion is the same as for the **"-g"** and **"-h"** command line options as well as for the BANKS.EXE linker. A file root name of no more than 5 characters is specified (the drive and full path can also be specified). These utilities look for all the files with the extensions **"_0", "_1"**, **"_2"**, and so forth up to a maximum of **"_63"**. Unlike the DOS COPY command, this utility will not overwrite a file. If any of the new files exist, none of the source files will be copied.

The default extension is **".lst"**, however any input or output extension can be specified. The extension can be changed by specifying a different output extension.

COPYLST Utility to copy a series of SBC source files - Ver 1.xx
Copyright 1994 P*ROM Software, Inc. - All Rights Reserved

Usage: copylst (srcrootname) (newrootname)

File root name can be up to 5 chars long. An underscore and bank number is automatically added to the old name and new name. E.g., the command:

copylst \sbc\spud \sbcprg\blue

will copy \sbc\spud_0.lst to \sbcprg\blue_0.lst, \sbc\spud_1.lst to \sbcprg\blue_1.lst, \sbc\spud_2.lst to \sbcprg\blue_2.lst, and so forth (up to 64 files).

The default extension is '.lst', but the extension can be specified for the source and/or new names.

RENLST.EXE

This utility renames all the files that have the specified root filename. Files can be renamed in the same directory, or can be "moved" to another directory by renaming them with a different path. Files cannot be renamed to another drive.

If any of the new files exists, none of the files will be renamed. The extension can be changed in the same manner as used for **COPYLST.EXE** above.

RENLST Utility to rename a series of SBC source files - Ver 1.xx
Copyright 1994 P*ROM Software, Inc. - All Rights Reserved

Usage: renlst (oldrootname) (newrootname)

File root name can be up to 5 chars long. An underscore and bank number is automatically added to the old name and new name. E.g., the command:

renlst \sbc\spud \sbcprg\blue

will rename \sbc\spud_0.lst to \sbcprg\blue_0.lst, \sbc\spud_1.lst to \sbcprg\blue_1.lst, \sbc\spud_2.lst to \sbcprg\blue_2.lst, and so forth (up to 64 files).

The default extension is '.lst', but the extension can be specified for the old and/or new names.

Files can be renamed to a different directory on the same drive but not from one drive to another.

Transferring Programs to Cassette Tapes and Other Calculators

By including a small routine in your program, you can use a PC-1270 to make a password-protected cassette tape of your program. The same procedure allows you to produce password-protected copies of your original program directly (without making a cassette tape) in other PC-1270 or PC-1250 series calculators. The program in the master PC-1270 must be unprotected (use the SBC switch option -U to produce an unprotected copy).

The following example is a program which will make password-protected copies of itself (provided it itself is NOT password protected) if the value entered in response to the "LOAN AMT?" prompt is -111.

```
LOAN_IN:
    INPUT "LOAN AMT? "; L :          ' If -111 entered, save
    IF L = -111                \     ' a protected copy of
      CSAVE, "XXX" :                 ' the program and
      GOTO LOAN_IN                   ' loop back for another

    INPUT "RATE? "; ...
```

If the entry is -111 AND the program in the originating calculator is NOT password-protected, the program will execute the **CSAVE** command and transmit a password-protected copy of the program to the cassette tape recorder (or another calculator through the EA-129C cable). The password is "XXX" (you can put any password you wish here, up to 7 bytes in length).

If the program in the originating calculator is password-protected, the **CSAVE** command is ignored and no program transmitted. (It is for this reason that a copy of a program made by following this procedure cannot itself generate additional program copies.)

The password itself only has significance for the PC-1250 series calculators and cassette tapes. If the program is transmitted to a PC-1250 series calculator, then the password specified in line 10 above will be the password installed in the PC-1250. Similarly, if the program is transmitted to a cassette tape and then a PC-1250 is made from the cassette tape, the password specified above will be installed in the PC-1250.

If the program is being transmitted to a PC-1270 (or transmitted to a cassette tape and then from the tape to a PC-1270), the password must be present above to tell the PC-1270 that the program is password protected. However, the particular password is immaterial because the PC-1270 does not actually store the password itself. There is no way for a user to remove the password in the PC-1270.

Entering Alpha Characters in the PC-1270

Often there is a need to enter alpha information on the PC-1270 and, because it has no alpha keyboard, it is somewhat difficult to do. The following routine shows a simple system using numbered codes for the letters of the alphabet. A is a 1, B is a 2, and so forth up to Z which is a 26. The space is 27, and the punctuation marks follow (see table below).

In this routine, the "A" function key displays the name if it has been installed, or displays the message "NO NAME INST" if no name has been installed. The "M" key is used to install a name. Each letter is entered by entering the 1 or 2 digit code number and pushing **[ENTER]**. If you make a mistake entering a character, you can push the "D" key to back up one letter.

```
'
'                    namestr.lst
'
'          To enter and display a "name" in the PC-1270
'
'
'
   ' Demo program showing installation of alphanumeric string
   ' from keyboard.
   '
   ' Key A displays the name string
   '
   ' Key D backup in input of string to correct an entry
   '
   ' Key M installs a new name string
   '
   '
   ' Characters are entered using code numbers from 1 thru 58 as
   ' shown in the following table.
   '
   '
   '   1 A     16 P     31 1     46 (
   '   2 B     17 Q     32 2     47 )
   '   3 C     18 R     33 3     48 +
   '   4 D     19 S     34 4     49 ,
   '   5 E     20 T     35 5     50 .
   '   6 F     21 U     36 6     51 /
   '   7 G     22 V     37 7     52 :
   '   8 H     23 W     38 8     53 ;
   '   9 I     24 X     39 9     54 <
   '  10 J     25 Y     40 !     55 =
   '  11 K     26 Z     41 "     56 >
   '  12 L     27 <space> 42 #     57 ?
   '  13 M     28 -     43 $     58 @
   '  14 N     29 *     44 %
   '  15 O     30 0     45 &

   ' variables used

   '   a   temporary integer
   '   b$  used in input prompts
   '   c   position pointer in alpha string
   '   z   initialized flag

   '   c$(0)   string of ok characters
   '   b$(0)   final alphanumeric string
```

```
    '   z$(9)   for flash card

    '
    ' back up one character position each time key D is
    ' pressed.  Can be used to correct alpha inputs.
    '

    "D"   \
    gosub Initialize :              ' check for initialization
    c = c - (c > 0) :               ' back up 1 char if on 1st+
    if c >= 0         \
      let b$(0) = left$(b$(0), c) : ' remove last character from string
      goto Char_input               ' go to input


    '
    '  display alphanumeric string, if it exists.
    '  If string does not exist, fall into input routine.
    '

    "A" \
    gosub Initialize :              ' check for initialization
    if asc b$(0) > 0    \           ' only print name if there is one.
      print b$(0) :
      end

    print "NO NAME INST" :
    end

    '
    '   input alphanumeric string from numeric pad
    '


    "M"   \
    gosub Initialize :          ' check for initialization
    pause "INSTALL NAME" :
    b$(0) = "" :                ' clear previous name
    c = 0                       ' clear position pointer

Char_input:
    a = 0 :                     ' clear temp input variable

    input ";b$(0);?"; a :
    a = int abs a               ' make sure input is a positive integer

    if a = 0   \                ' if input = 0, quit
      goto No_more_chars

    if a <= 58   \              ' concatenate alpha string
      let b$(0) = b$(0) + mid$(c$(0), a, 1) :
      c = c + 1 :               ' increment position pointer
      if c = 16 \               ' no more char space available
        goto No_more_chars

    goto Char_input             ' loop back for next character

No_more_chars:
    c = 0 :                     ' disable backup key when done
    print b$(0) :               ' display entered name
```

```
    end


    '
    '  initialize subroutine
    '
Initialize:

    if z  \
      return                    ' already initialized

    clear :
    dim z$(9) * 1,     \            ' for bank switching
       c$(0) * 58,    \            ' list of usable characters
       b$(0) * 16 :                ' alphanumeric string


    '  NB: quotation mark must be placed in string by
    '  chr$(34) & semi-colon (";") by chr$(59).

    c$(0) = "ABCDEFGHIJKLMNOPQRSTUVWXYZ -*0123456789!" \
          + chr$(34) + "#$%&()+,./:" + chr$(59) + "<=>?@" :

    z = 1 :                     ' set initialized  flag
    return
```

When the above source file is decompiled, the actual code the PC-1270 will execute is produced as follows:

```
 1 "D" GOSUB 10 : C=C-(C>0) : IF C>=0 LET B$(0)= LEFT$ (B$(0),C) : GOTO 5
 2 "A" GOSUB 10 : IF ASC B$(0)>0 PRINT B$(0) : END
 3 PRINT "NO NAME INST" : END
 4 "M" GOSUB 10 : PAUSE "INSTALL NAME" : B$(0)="" : C=0
 5 A=0 : INPUT ";B$(0);?";A : A= INT ABS A
 6 IF A=0 GOTO 9
 7 IF A<=58 LET B$(0)=B$(0)+ MID$ (C$(0),A,1) :
     C=C+1 : IF C=16 GOTO 9
 8 GOTO 5
 9 C=0 : PRINT B$(0) : END
10 IF Z RETURN
11 CLEAR : DIM Z$(9)*1,C$(0)*58,B$(0)*16 :
     C$(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ -*0123456789!"+ \
     CHR$ (34)+"#$%&()+./:"+CHR$ (59)+"<=>?@" : Z=1 : RETURN
```