Remember that a variable is a box into which a number or a character string is stored and from which it is taken out when needed. DIM A(9) in line 20 of the sample program prepares a row of 10 boxes. These boxes are labeled A(0), A(1), . . . , A(9).

To set up five boxes to store character strings, use the statement DIM A$(4). That command will prepare boxes A$(0) through A$(4). This row of boxes is called an array.

The utility of arrays may not be obvious in a small program. In larger programs doing more complex tasks, however, grouping variables into arrays will be very handy. To do that, though, you need to learn the commands frequently used with arrays.

●Sample Program 8

```
10 CLEAR
20 DIM A$(2)
30 FOR I=0 TO 2
40 READ A$(I)
50 PRINT A$(I)
60 NEXT
70 END
100 DATA SEIKO,UC,-2200
```

When you run this program, the screen will display SEIKO, UC, and −2200 on separate lines. Line 20 in the program is a DIM statement which defines an array. Here the array A$(2) is defined, and the computer sets up three boxes to hold character strings. Lines 40 and 100 work together.

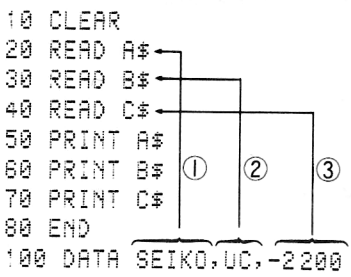
```
 40 READ A$(I)
100 DATA SEIKO,UC,-2200
```

READ and DATA statements are often used with arrays. READ and DATA are never found independently, but always form a pair.

The READ statement tells the UC-2200 to read something into its memory. READ A$(I) tells it to read string data into the variable A$(I).

DATA has data to be read in by READ. If more than one item is to be read in, separate them with commas in the DATA statement.

The order in which READ statements have the data read in:



The program above was written without using an array. The DATA statement gives three pieces of data, separated by commas from each other. But this statement does not tell us which datum should be assigned to which variable. The rule is simple, though: the first READ statement reads the first item in the DATA statement, the second reads the second item, and so on. Thus the character string SEIKO, the first item in the DATA statement, goes into the variable A$; UC goes to B$; and −2200 to C$. Your UC-2200 remembers that it read in SEIKO first and always gives the variable in the second READ statement the UC it read in second. The data in this example were character strings. In the DATA statement, however, string data need not be enclosed in double quotation marks. String data and numerical data are handled in the same way in DATA statements. What's more, numerical and string data can share the same DATA statement. Remember, though, that the READ statements must distinguish between string and numerical variables.

The RESTORE command is sometimes used with READ and DATA statements. The RESTORE command is used in programs with several DATA statements. It has the computer read in the data in the DATA statements once again, after the READ statement has been executed once. The RESTORE statement is executed, the READ statement is executed again, and the data are read in, so that the variables are given the same values as before. If a line number is given after RESTORE (RESTORE 20), the reading in starts again from the DATA statement at the line number given.

●Sample Program 9

```
10 CLEAR:DIM A$(3,4)
20 FOR I=1 TO 3:FOR
J=1 TO 4
30 READ A$(I,J)
40 NEXT J,I:CLS
50 INPUT "WHAT COLUM
N";A
60 INPUT "WHAT ROW";
B
70 PRINT A;"COLUMN";
B;"ROW"
80 PRINT A$(A,B):PRI
NT
90 GOTO 50
200 DATA TOM,JOHN,SU
SAN,PEARL
210 DATA LINDA,OLIVI
A,RICHARD,DAVID
220 DATA JERRY,MICHA
EL,TERRY,ROBERT
```

This program makes use of DIM, READ, and DATA statements. These should be familiar to you now. In this program, however, the array defined by the DIM statement is a little different. Look at line 10 in the sample program. It has two numbers in parentheses after the DIM A$ statement

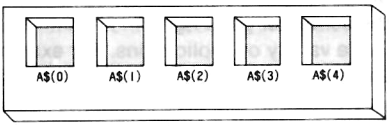
A$ (3, 4)

Sample Program 7 had one number in parentheses after the name of the array in its DIM statement. That number is called a subscript. Here, however, we have two numbers or subscripts separated by a comma. An array defined with one subscript is a one-dimensional array. An array with two subscripts is a two-dimensional array.

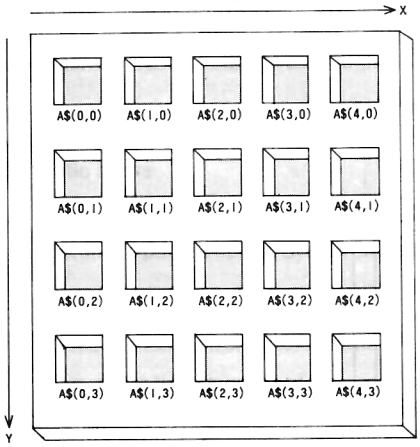
The one-dimensional array in Sample Program 7 could be thought of as a row of boxes, in which each box was a variable. With a two-dimensional array, we have both rows and columns of boxes. Figure A shows the difference between an one-dimensional and a two-dimensional array.

Figure A A group of variables defined by an array

One-dimensional array: DIM A$(4)

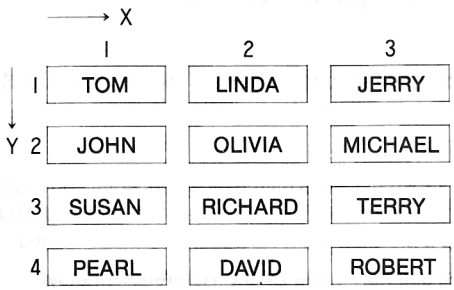


Two-dimensional array: DIM A$(4,3)



To set up a two-dimensional array, all you need to do is input numbers for horizontal(X) and vertical (Y)coordinates: DIM A$(X, Y). As in one-dimensional arrays, the boxes can be filled by character strings or numbers. To designate a datum to be stored or fetched, it is only necessary to specify the location of its box according to the X-Y coordinate system. With a single row of boxes, a single ordinal number did the job of specifying a particular box. But to identify a particular box in a two-dimensional array, you need to give two numbers, its ordinal position in its row and its row number.

Figure B Arrangement of Desks

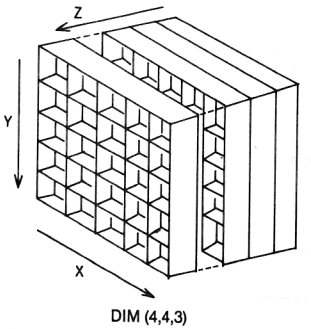


The sample program is designed to check what student has what place in a classroom with 12 desks arranged in three columns and four rows. The names of the students and their places are shown in Figure B. Run the program, key in the coordinates, and check the names of the students displayed. Did the computer get it right?

Two-dimensional arrays have a wide variety of applications. For example, you could use a two-dimensional array to write an Othello game program. Arrays are definitely a convenient way to handle variables.

You can define higher-order arrays, too — three, four, or more dimensions. A three-dimensional array can be thought of as a solid block built of boxes (Figure C). The problem is that it is hard to think about operations involving the higher dimensions. Two-dimensional arrays answer most of our requirements.

Figure C A Three-Dimensional Array



5-3 Other BASIC Commands

In addition to the commands introduced so far, your UC-2200 can respond to some special commands which are not available in other versions of BASIC.

●MPRINT

MPRINT works in the same way as PRINT and LPRINT, but output is sent to the memory of the UC-2000 instead of to the screen or the printer. Executing an MPRINT statement has a memo or data from the UC-2200 stored in the memory (Memory B) of the watch.

The first MPRINT statement executed clears Memo B. MPRINT can be used to store a maximum of 1000 characters (the capacity of Memo B in the watch).

●STOP

This command makes the program stop at the line number in which the STOP occurs. The screen will display the cursor in readiness for a command or an input.

●CONT

CONT unlocks the program after it has been halted by STOP. But if you modified the program after the STOP command, you cannot use CONT to restart it. CONT and STOP are convenient for checking and debugging a program which is not working as planned.

For other BASIC commands, see Part II, the BASIC Reference Manual.

5-4 Intrinsic Functions

UC-2200 BASIC has several intrinsic functions available. They include trigonometric, logarithmic, string, and random number functions. And you can define other functions as you need them. Greater confidence as a programmer will rest on mastery of these intrinsic functions.

Chapter 2 of Part II explains the intrinsic functions available in UC-2200 BASIC. Read it carefully remember what intrinsic functions are available.

5-5 Coping with Error Messages

Once in a while you will find an error when you try to run a program. Naturally, you will check through your program to debug it. Errors are roughly classifiable into three types:

1. Syntax errors
2. Numerical errors
3. Semantic errors

- Syntax errors are usually caused by minor mistakes — entry of the wrong letter from the keyboard or forgetting to put in a space. Syntax errors are easy to correct, once you spot the error.
- Numerical errors are caused by exceeding the numerical capacity of the UC-2200 during computation. Correcting such errors takes a little more effort.
- Semantic errors are errors that do not produce error messages. That is, the UC-2200 has not found a syntactic or numeric error, but the program does not do what you expected. These errors are the hardest to correct.

●Syntax Errors

When a syntax error occurs, the program stops running, and the screen displays the number of the line in which the error was detected. Press the F2 key or key in E D T , then key in the number of the line given in the error message, and press RETURN . The computer will enter EDIT mode and display the designated line. Use the cursor keys to move the cursor to the trouble spot and correct it. Then press RETURN . Your error is correced.

Most syntax errors are the result of typographical errors, leaving out a space, or forgetting to key in the sign. You can usually detect syntax errors by reading over the program listing. The most common errors are typographical.

In addition, syntax errors are caused by not following the rules for formatting statements. For instance, omitting the line number to which the program is to jump from a GOTO statement or a GOSUB statement or trying to call up a function not defined by DEFFN will result in an error message. These syntax errors are easy to spot. When you get a syntax error, the screen or printer will give you an error message. Refer to Appendix 2, Error Messages, to determine what the error is and how to correct it.

●Numerical Errors

Numerical errors also lead to error messages. That helps with correction, but if the error is due to the value of a variable, you must check through all the lines in which that variable occurs. Debugging numerical errors takes patience.

Numerical errors occur when the value of the variable or expression used in the program exceeds the memory capacity of the UC-2200. But there are other possible sources of numerical errors — trying to divide by zero in expressions such as A = B/C, for instance. That gives you an error message, /0 ERROR on the screen, making the bug simple to detect. But variable-tied errors can be harder to correct, because you have to check every line in which that variable occurs.

Once you have tracked down the source of a numerical error, correct it just as you would a syntax error. You may have to add a new line or delete a line from your program.

●Semantic Errors

Semantic errors are the most recalcitrant. The UC-2200 does just what its program tells it to.

If there are no bugs in the program, your computer runs it, no matter if the data are wrong or the variables have the wrong values. It gives the answer to what you asked it, not what you should have asked it. Debugging this kind of error calls for much thought.

There are too many ways to create semantic errors to give an exhaustive list of corrective methods for them. But the overwhelming majority of semantic errors are linked to variables.

For example, a semantic error occurs when the value of a variable is destroyed or when an erroneous statement assigns a wrong value to a variable. These errors are especially treacherous if the variable is used in IF, ON-GOTO, or ON-GOSUB statements. These statements shunt the flow of the program all over, depending on the value the variable takes. You must work through all the branches of the program in search of bugs.

To prevent this situation, try checking the values of the variables in advance by inserting STOP statements at strategic places in the program. During processing, the program will pause whenever it comes to a STOP statement, allowing you to check the values of the variables efficiently. Then you can catch semantic errors in action. It is advisable to run through the program with STOP-CONT statements repeatedly to be sure none of your variables get out of line. And after you have found and corrected a bug, check the program again. The longer the program, the more trouble debugging can be.

A comprehensive table of variable names and values will be a great help in correcting semantic errors. This table, a table of variables, is indispensable not only for debugging but also for writing programs. Form the habit of drawing one up for each of your programs.

PART II BASIC Reference Manual

This manual covers all BASIC commands/statements and intrinsic functions available on the UC-2200.

Chapter 1 Commands/Statements

In this chapter, commands or statements are presented in the following form:

Format ..... Shows the correct format for the command/statement. See below for format notation.
Purpose ..... Explains what the command/statement is used for.
Remarks ..... Describes in detail how to use the command/statement.
Examples ..... Gives sample programs to demonstrate the use of the command/statement.

FORMAT NOTATION

The following rules apply to the format for a statement or command:

1. Items in capital letters must be input exactly as shown.
2. Items in lower-case letters enclosed in angle brackets < > must be supplied by the programmer.
3. Items in square brackets [ ] are optional.
4. All punctuation marks except square and angle brackets — commas, parentheses, semicolons, hyphens, equal signs — must be used where shown.
5. Items followed by ellipses . . . may be repeated any of number of times, up to the length of a logical line.
6. Items between vertical lines || are to be selected by the user.

BEEP

Format BEEP
Purpose To produce a beeping sound

Remarks The beep will continue for about one second, then stop automatically.

Example

```
10 BEEP
20 FOR I=1 TO 10
30 BEEP
40 NEXT I
50 END
```

CLEAR

Format CLEAR [<expression 1>]
Purpose To set all numeric variables to zero, all string variables to null.

Remarks <expression 1> is a number which reserves space in the UC-2200 memory for working with strings. If not specified, the computer reserves 50 bytes for strings.

Example

```
CLEAR
```

CLS (Clear Screen)

Format CLS
Purpose To clear all the character and graphics from the screen.

Remarks The cursor moves to home position after CLS is executed.

Example

```
10 CLS
```

CONT (Continue)

Format CONT

Purpose To resume program execution after the STOP key has been pressed or a STOP statement has been executed.

Remarks Execution resumes where the break occurred. CONT cannot be used if you edited the program during the break. If the break occurred after a prompt from an INPUT statement, execution resumes with the re-display of the prompt (a question mark or string and question mark). CONT is usually used with STOP for debugging. You can check intermediate values of variables while the program has paused, using direct mode commands such as PRINT. Execution can be resumed with CONT or with a direct mode GOTO, which has the program resume execution at the line number specified.

Example See STOP

DATA

Format DATA <constants> [, <constants>]...

Purpose To store numeric and string constants accessed by the program's READ statement.

Remarks DATA statements are always used with READ statements. DATA statements can be placed anywhere in the program. A DATA statement may contain as many constants (separated by commas) as will fit in a logical line. A program can contain any number of DATA statements.
READ statements access the DATA statements in order (by line number), and the data in them forms a single continuous list of items. It does not matter how many items are in each DATA statement or where the DATA statements have been placed in the program; the READ statements have the items in the DATA statements read in one by one, in the order in which they occur.

Example

```
10 READ A
20 READ B$
30 READ C$
40 PRINT A,B$,C$
50 DATA 2200,SEIKO,2
200

RUN
 2200 SEIKO 2200
OK
```

DEF FN (Define function)

Format DEF FN <name> [(<parameter>)] = <function definition>
Purpose To define a function written by the user.

Remarks

<name> must be a legal variable name. This name, preceded by FN, becomes the name of the user-defined function.
<parameter> consists of the variables in the <function definition> that will be replaced when the function is called.
<function definition> is the operation that the function performs. It is limited to one logical line. Variables that appear in the <function definition> may or may not appear in <parameter>. If a variable does appear in <parameter>, the value of the <parameter> is supplied when the function is called. Otherwise, the current value of the variable is used.
A DEF FN statement must be executed before the program calls the function it defines. Calling the function before it has been defined leads to a UF Error.
DEF FN cannot be used in direct mode.
In UC-2200 BASIC, only numeric user-defined functions are allowed.

Examle

```
410 DEF FNAB(X )=X^3
420 T=FNAB(I )
```

Line 410 defines the function FNAB.
Line 420 calls the function.

DIM (Dimensions)

Format DIM <subscripted variables>
Purpose To specify the maximum value for array variable subscripts and allocate storage accordingly.

Remarks

If an array variable is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10.
Using a subscript greater than the maximum specified leads to a BS Error.
The minimum value for a subscript is always 0.
The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
```

EDIT

Format EDIT <line number>
Purpose To edit the program at the specified line.

Remarks In EDIT mode, you can edit portions of lines without retyping the entire line. On the UC-2200, EDIT has been assigned to function key F2 . See Chapter 3, The Function Keys.

Example

```
EDIT 10
10 CLEAR
```

END

Format END
Purpose To terminate program execution, close all files, and return to command level.

Remarks END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK statement to be displayed. BASIC always returns to command level after an END statement is executed.

Example

```
520 IF K>1000 THEN E
ND
```

FOR-NEXT STEP

Format FOR <variable> = X TO Y [STEP Z]

NEXT [<variable>] [,<variable>]...

Purpose To have a series of instructions performed in a loop a given number of times.

Remarks <variable> is used as a counter. The first number (X) is the initial value of the counter, and the second number (Y) is its final value. (X, Y, and Z are all numbers.) The program lines following the FOR statement are excuted until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. The computer checks to see whether the value of the counter is now greater than the final value (Y). If not, it branches back to the statement after the FOR statement and repeats the process. If the counter is greater, the computer goes on to execute the statement following the NEXT statement. This set of commands is called a FOR-NEXT loop.

If STEP is not specified, the increment is assumed to be one.

If STEP is negative, the final value of the counter must be less than the initial value.

Example

```
10 FOR I=0 TO 20 STE
P 5
20 A=10*I
30 PRINT A
40 NEXT I
50 END
```

GOSUB-RETURN

Format GOSUB <line number>
⋮
RETURN

Purpose To branch to and return from a subroutine.

Remarks <line number> is the first line of the subroutine.
A subroutine may be called any number of times in a program. A subroutine may also be called from within another subroutine. The only limit on such nesting of subroutines is the memory available.
The return statement in a subroutine has the computer branch back to the statement following the nearest GOSUB statement.

Example

```
10 PRINT "SEIKO"
20 GOSUB 50
30 PRINT "2200"
40 PRINT "END":END
50 PRINT "UC-"
60 RETURN

RUN
SEIKO
UC-
2200
END
OK
```

GOTO

Format GOTO <line number>

Purpose To branch unconditionally out of the normal program sequence to the specified line number.

Remarks The computer will execute the statement at the specified line number and those following it.

Example

```
10 READ R
20 PRINT "R =";R
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12

RUN
R = 5
AREA =        78.5
R = 7
AREA =        153.86
R = 12
AREA =        452.16
OK
```

IF ~ THEN, IF ~ GOTO

Format 1 IF <expression> THEN <instruction>|<line number>
Format 2 IF <expression> GOTO <line number>
Purpose To make a decision about program flow based on whether the <expression> is true.

Remarks If the <expression> is true, the second clause is put into effect, with the result depending on the format used:

Format 1: Executes the instruction following the THEN clause or branches to the specified line number.

Format 2: Executes the GOTO clause and branches to the specified line number.

If the <expression> is false, the THEN or GOTO clause is ignored, and the next program line is executed.

Example

```
10 INPUT A
20 IF A=0 THEN PRINT
 "ZERO":GOTO 10
30 IF A<0 GOTO 50
40 IF A>0 GOTO 60
50 PRINT "NEGATIVE":
GOTO 10
60 PRINT "POSITIVE":
GOTO 10

RUN
? 5
POSITIVE
? -5
NEGATIVE
? 0
ZERO
```

INPUT

Format INPUT [<"prompt string">;] <variable> [,<variable>]...
Purpose To allow input from keyboard during program execution.

Remarks

When the program reaches an INPUT statement, the computer pauses and displays a question mark to indicate that it is waiting for data. If <"prompt string"> was included in the INPUT statement, the string is displayed before the question mark. Enter the required data from the keyboard. The data you enter are assigned to the variable (s) listed in <variable>. You must supply the same number of data items as the number of variables listed. Data items must be separated by commas. Variable names given in <variable> may be numeric or string, and may be subscripted variables. Each data item input must agree with the type specified by the variable name. String data input in response to an INPUT statement, however, need not be surrounded by quotation marks.

Responding to an INPUT command with too many or too few items or with the wroing type of data (numeric instead of string, for instance) causes an ID error. The computer will not assign input values and go on with the program execution until you have given an acceptable response to the INPUT command.

Example:

```
10 INPUT A
20 PRINT A "SQUARED
IS" A^2
30 END


RUN
? 5
 5 SQUARED IS 25
OK
```

LET

Format LET <variable> = <expression>
Purpose To assign the value of an expression to a variable.

Remarks LET is an optional command. The equals sign is sufficient to assign an expression to a variable name.

Example

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
```

or

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
```

LIST

Format LIST [<line number>]
Purpose To list on the screen all or part of the program currently in memory.

Remarks BASIC always returns to command level after LIST is executed.
If the <line number> is omitted, the whole program is listed.
If the <line number> is included, the program is listed from the specified line number.

Example

```
LIST
LIST 500
```

LLIST

Format LLIST [<line number>]
Purpose To print out all or part of the program currently in memory.

Remarks BASIC always returns to command level after LLIST.
The options for LLIST are the same as for LIST.

Example

```
LLIST
LLIST 500
```

LOCATE

Format LOCATE <horizontal position>, <vertical position>
Purpose To move the cursor to the specified position on the screen.

Remarks The value of <horizontal position> must be from 0 to 9 and <vertical position> must be from 0 to 3.

Example

```
10 CLS
20 LOCATE 2,1:PRINT
"SEIKO"
30 LOCATE 2,2:PRINT
"UC-2200"
40 GOTO 10
```

LPRINT

Format LPRINT [<expression> [;[<expression>]]...]
[,[<expression>]]...]

Purpose To print out data with the printer

Remarks Same as PRINT except that output goes to the printer.

Print Positions

UC-2200 BASIC divides the line into printing zones of 8 spaces each. In the list of <expression>, a comma makes the next value be printed at the beginning of the next zone.

A semicolon makes the next value be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing in a semicolon.

If the list of <expression> ends with a comma, the next LPRINT statement begins printing on the same line, spacing accordingly.

If a list of <expression> ends without a comma or semicolon, a carriage return is inserted at the end of the line and the next LPRINT statement begins printing at the beginning of the next line.

If the line to be printed is longer than the printer width (20 characters), BASIC wraps around to the next line and continues printing.

MPRINT

Format MPRINT [<expression> [;[<expression>]]...]
[,[<expression>]]...]

Purpose To store data from a program in Memo B of the UC-2000 watch.

Remarks Works as PRINT does expect output goes to Memo B of the watch. Please note the following points:

1. The first execution of an MPRINT statement after entering BASIC mode will clear Memo B and begin storing data at the start of Memo B. Subsequent MPRINT statements will store data further down in Memo B. But if you leave BASIC mode, return to it, and again execute an MPRINT statement, that data will be stored at the beginning of Memo B and any data stored earlier will be erased.
2. If your schedule or a game program has already been transferred to the watch, the first execution of an MPRINT statement will reset the watch to Memo mode, clearing the program, and will initialize memo storage.
3. A maximum of 1000 characters can be stored in Memo B. If that maximum is exceeded, the printer will print out a Memory Limit Error message.

NEW

Format NEW

Purpose To delete the program currently in memory and clear all variables.

Remarks NEW is used at command level to clear memory before entering a new program. BASIC returns to command level after executing a NEW command.

ON-GOSUB, ON-GOTO

Format ON <expression> GOSUB <list of line numbers>
ON <expression> GOTO <list of line numbers>

Purpose To branch to one of several specified line numbers, depending on the value returned when the expression is evaluated.

Remarks The value of <expression> determines which line number in the list the program will branch to. For example, if <expression> equals three, the third line number in the list will be the destination of the branch. If the value is not an integer, the factional portion is dropped.
In an ON-GOSUB statement, each line number in the list must be the first line of a subroutine.
If the value of the <expression> is zero or greater than the number of line numbers in the list (but less than or equal to 255), the program goes on to the next executable statement. If the value of <expression> is negative or greater than 255, an FC error occurs.

Example

```
100 ON L-1 GOTO 150,
300,320,390
```

PRINT

Format PRINT [<expression>][;[<expression>]]...]
[,[<expression>]]...]

Purpose To display data on the screen

Remarks If <expression> is included, the expression is displayed on the screen. It may be a numeric or string expression. Strings must be in quotation marks.

Print Positions

The position of each item is determined by the punctuation used to separate the items in the list of <expression>.

A comma makes the next value be displayed at the beginning of the next line.

A semicolon makes the next value be displayed immediately after the last one. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or semicolon terminates the list of <expression>, the next PRINT statement begins display on the same line.

If the list of <expression> ends without a comma or semicolon, a carriage return is assumed at the end of the line and the next PRINT statement begins display at the beginning of the next line.

If the line to be displayed is longer than the screen width (10 characters), the program goes to the next physical line and continues displaying.

Example

```
10 INPUT A
20 PRINT A;
30 PRINT " SQUARED"
40 PRINT
50 PRINT "IS";A^2
60 GOTO 10

OK
RUN
? 5
 5 SQUARED

IS 25
```

READ

Format READ <list of variables>
Purpose To read values from a DATA statement and assign them to variables.

Remarks READ statements must be used with DATA statements. The READ statement assigns variables to DATA statement values on a one-to-one basis, in the order in which they occur.
READ statement variables may be numeric or string, but the values in the DATA statement must agree with the variable types specified in the READ statements. If they do not agree, a SN error occurs.
A single READ statement may access one or more DATA statements, in the order in which they occur. Or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statements, an OD error occurs. If the number of variables specified is fewer than the number of elements in the DATA statement, subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra elements are ignored.
To reread DATA statements from the start, use RESTORE.
(See RESTORE.)
This program segment READs the values from the DATA statements into the array A. A(1) will have the value of 5, and so on.

Example

```
10 DIM A(5)
20 FOR I=1 TO 5
30 READ A(I)
40 NEXT I
50 DATA 5,4,3,2,1
60 FOR I=1 TO 5
70 PRINT "A(";I;")="
;A(I)
80 NEXT I
```

REM (Remark)

Format REM <remark>
Purpose To allow you to insert explanatory remarks into your program.

Remarks REM statements are not executed but are output exactly as entered when the program is listed.
REM statements may be branched to from a GOTO or GOSUB statement. Execution will continue with the first executable statement after the REM statement.

Example

```
100 REM UC-2200 MEMO
```

RESTORE

Format RESTORE [<line number>]
Purpose To allow DATA statements to be reread when processing reaches a specified line.

Remarks After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program.
If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example

```
10 READ A$,B$,C$
20 K$=A$+B$+C$
30 PRINT K$
40 DATA "X","Y","Z"
50 RESTORE 40
60 READ D$,E$,F$
70 L$=F$+E$+D$
80 PRINT L$

RUN
 XYZ
 ZYX
OK
```

RUN

Format RUN [<line number>]
Purpose To execute the program currently in memory.

Remarks If <line number> is specified, execution begins on that line. Otherwise, it begins at the first line. BASIC always returns to the command level after a RUN statement is executed.

Example

```
RUN

RUN 100
```

STOP

Format STOP

Purpose To halt program execution and return to command level.

Remarks The STOP statement may be used anywhere in a program to insert a pause in execution. When a STOP is encountered, the screen displays the following message:

Break in nnnn (nnnn is the line number)

BASIC always returns to the command level after a STOP command. Resume execution by issuing a CONT statement.
(See CONT.)

Example

```
10 INPUT C,D,E
20 K=C*D*E:L=C+D+E
30 STOP
40 M=K-L:PRINT M
50 GOTO 10

RUN
? 5,4,3
Break in 30
OK
PRINT K
 60
OK
CONT
 48
?
```

Chapter 2 Intrinsic Functions

This chapter describes all the intrinsic functions built into UC-2200 BASIC. Each description is in the following form:

Format .....Shows the correct format for the function.
Action .....Explains what the function does.
Example .....Gives sample programs demonstrating the use of the function.

FORMAT NOTATION

Since they are a part of UC-2200 BASIC, the intrinsic functions may be called from any program without further definition.

The expressions to which the function is applies are always enclosed in parentheses. In this chapter, those expressions have been named as follows:

X and Y	Numeric expressions
I and J	Integer expressions
X$ and Y$	String expressions

If a floating point value is supplied where an integer is required, BASIC will drop the fraction and use the resulting integer. Functions return only integer and single precision results.

ABS

Format ABS(X)
Action Finds the absolute value of X

Example

```
10 INPUT A
20 PRINT ABS(2*A)
30 END

RUN
? -15
 30
OK
```

AND

See the memo on logical operators.

ASC (ASCII)

Format ASC (X$)

Action Finds the number that is the ASCII code for the first character of the string X$. See CHR$ for ASCII-to-string conversion.

Example

```
10 X$= "UC-2200"
20 PRINT ASC(X$)

RUN
 85
OK
```

ATN (arctangent)

Format ATN (X)

Action Calculates the arctangent of X in radians. Result is in the range — $\pi/2$ to $\pi/2$.

Example

```
10 INPUT X
20 PRINT ATN(X)

RUN
? 3
 1.24905
OK
```

CHR$

Format CHR$ (I)

Action Converts an ASCII code to the character it represents. CHR$ is commonly used to send a special character to the screen. For instance, the BEEP character could be sent CHR$(7) as a preface to an error message, or CHR$ (12) could be sent to clear the screen and return the cursor to home.

Example

```
PRINT CHR$(66)
B
OK
```

COS (Cosine)

Format COS (X)
Action Calculates the cosine of X in radians.

Example

```
10 PRINT COS(3.14159
/3)

RUN
 .500001
OK
```

CSRLIN (Cursor line)

Format CSRLIN
Action Gives the current line number of the cursor location

Example

```
PRINT CSRLIN
 3
OK
```

EXP (Exponent)

Format EXP (X)
Action Calculates e to the power of X. (X must be < = 87.3365.) If EXP overflows available memory, an OV Error occurs, the largest possible value with the appropriate sign is supplied as the result, and execution continues.

Example

```
10 X=5
20 PRINT EXP(X-1)
RUN
 54.5982
OK
```

FRE (Free)

Format FRE (0)
FRE (X$)

Action FRE (0) tells you the number of bytes free in user-area memory.
FRE (X$) tells you the number of bytes free in the string area.

Example

```
PRINT FRE(0)
 2892
OK
```

INT (Integer)

Format INT (X)

Action Returns the largest integer not greater than X.

Example

```
PRINT INT(99.89)
 99
OK

PRINT INT(-12.11)
-13
OK
```

LEFT$

Format LEFT$ (X$, I)

Action Returns a string consisting of the leftmost I characters of X$, where I is from 0 to 255. If I is greater than the number of characters in X$, the result returned will be the entire X$ string.
If I = 0, the null string (length zero) is returned. See also MID$ and RIGHT$.

Example

```
10 A$ = "SEIKO-UC"
20 B$ = LEFT$(A$,2)
30 PRINT B$
```

LEN

Format LEN (X$)
Action Gives the number of characters in the string X$. Counts non-printing characters and blanks.

Example

```
10 X$ = "NEW YORK &
"TOKYO"
20 PRINT LEN(X$)

RUN
 15
OK
```

LOG (Logarithm)

Format LOG (X)
Action Calculates the natural logarithm of X. X must be greater than zero.

Example

```
PRINT LOG(45/7)
 1.86075
OK
```

MID$

Format MID$ (X$, I [,J])

Action Returns a string of length J, beginning with the Ith character of X$. I and J must be between 1 and 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all characters from the Ith character to the right end of the string will be returned.
If I is greater than the number of characters in X$, MID$ returns a null string.
See also LEFT$ AND RIGHT$.

Examples

```
10 A$="GOOD "
20 B$="MORNING EVENI
NG AFTERNOON"
30 PRINT A$
40 PRINT MID$(B$,9,7
)

RUN
GOOD
EVENING
OK
```

NOT

See the memo on logical operators.

OR

See the memo on logical operators.

POS

Format POS (I)

Action Tells you the present horizontal position of the cursor. The leftmost position is 1. (I) is a dummy variable.

Example

```
IF POS(X)>5 THEN PRI
NT CHR$(13)
```

RIGHT$

Format RIGHT$ (X$, I)

Action Returns the rightmost I characters of string X$.If the number of characters in X$ is I, RIGHT$ returns X$. If I = 0, the null string (length zero) is returned. See also MID$ and LEFT$.

Example

```
10 A$="UC BASIC"
20 PRINT RIGHT$(A$,5
)

RUN
BASIC
OK
```

RND (Random)

Format RND (X)

Action Returns a random number between 0 and 1.
The same sequence of random numbers is generated each the program runs unless the random number is reseeded. But if X is less than 0, RND always restarts the same sequence for any given X. If X is greater than 0, RND generates the next random number in the sequence.
If X = 0, RND repeats the last number generated.

Example

```
10 FOR I=1 TO 5
20 PRINT INT(RND(I)*
100);
30 NEXT I

RUN
 24 30 31 51 5
OK
```

SGN (Sign)

Format SGN (X)
Action If X>0, SGN (X) is 1.
If X=0, SGN (X) is 0.
If X<0, SGN (X) is −1.

Example

```
ON SGN(X)+2 GOTO 100
,200,300
```

If X is negative, branches to line 100.
If X is positive, branches to line 200.
If X is zero, branches to line 300.

SIN (Sine)

Format SIN (X)
Action Calculates the sine of X in radians.

Example

```
PRINT SIN(3.14159/6)
 .5
OK
```

SPC

Format SPC (I)
Action Outputs I blanks on the screen, printer, or Memo B. Can only be used with PRINT, LPRINT, or MPRINT statements.
I must be from 0 to 255.

Example

```
10 LPRINT "SPACE" SP
C(5) "FIVE"

RUN
SPACE     FIVE
OK
```

SQR (Square root)

Format SQR (X)
Action Returns the square root of X. X must be > = 0.

Example

```
10 FOR X = 10 TO 25
STEP 5
20 PRINT X, SQR(X)
30 NEXT X

RUN
 10
 3.16228
 15
 3.87298
 20
 4.47214
 25        5
OK
```

STR$

Format STR$ (X)
Action Converts a number (X) to a string. See also VAL.

Example

```
10 INPUT N
20 ON LEN(STR$(N)) G
OSUB 100,200,300
```

TAB

Format TAB (I)

Action Spaces to position I on the output device (screen or printer). If the current print or display position is already beyond space I, TAB goes to that position on the next line.

I must be between 1 and 255.

TAB may be used only in PRINT, LPRINT, and MPRINT statements.

Example

```
10 PRINT "ODD" TAB(5
) "EVEN":PRINT
20 FOR I=1 TO 3
30 READ A,B
40 PRINT A TAB(5) B
50 NEXT I
60 DATA 1,2,3,4,5,6

RUN
ODD   EVEN

 1     2
 3     4
 5     6
OK
```

TAN (Tangent)

Format TAN (X)

Action Calculates the tangent of X in radians.

If TAN overflows available memory, an OV error occurs, the largest available number with the appropriate sign will be supplied as the result, and execution continues.

Example

```
10 PRINT TAN(3.14159
/4)


RUN
 .999999
OK
```

VAL (Value)

Format VAL (X$)

Action Converts a string (X$) to a number. See also STR$ for numeric to string conversion.

Example

```
10 PRINT VAL("2200")
20 PRINT VAL("UC-220
0")
30 PRINT VAL("2200-U
C")

RUN
 2200
 0
 2200
OK
```

Appendix I

List of Commands/Statements and Intrinsic Functions

Command/Statement	Intrinsic Function
BEEP	ABS
CLEAR	AND
CLS	ASC
CONT	ATN
DATA	CHR$
DEF FN	COS
DIM	CSRLIN
EDIT	EXP
END	FRE
FOR~NEXT	INT
GOSUB~RETURN	LEFT$
GOTO	LEN
IF~THEN, IF~GOTO	LOG
INPUT	MID$
LET	NOT
LIST	OR
LLIST	POS
LOCATE	RIGHT$
LPRINT	RND
MPRINT	SGN
NEW	SIN
ON~GOSUB, ON~GOTO	SPC
PRINT	SQR
READ	STR$
REM	TAB
RESTORE	TAN
RUN	VAL
STOP	

Appendix 2

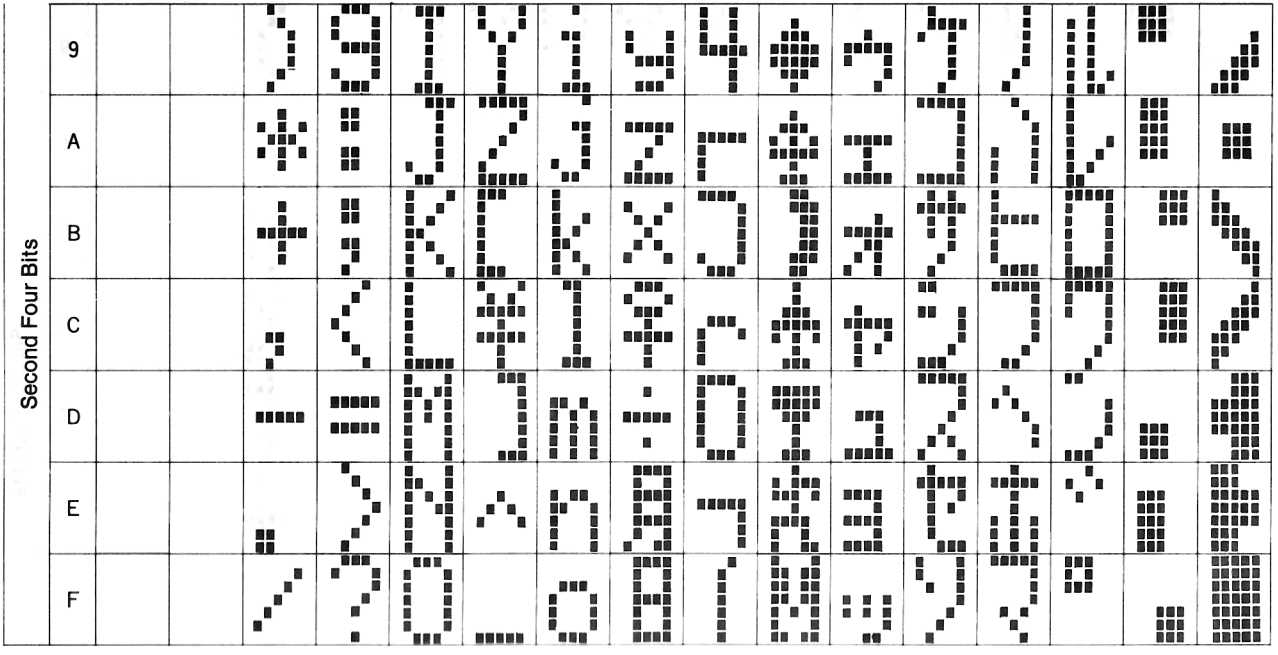
Error Messages

Code	Message
NF	**NEXT without FOR** A variable in a NEXT statement does not correspond to a variable in any previously executed, unmatched FOR statement.
SN	**Syntax error** A line is encountered that contains incorrect sequence of characters. (unmatched parenthesis, misspelled command/statement, incorrect punctuation).
RG	**RETURN without GOSUB** A RETURN statement is encountered for which there is no previous unmatched GOSUB statement.
OD	**Out of data** A READ statements executed when there are no DATA statements with unread data left.
FC	**Illegal function call** A parameter that is out of range is used with an arithmetical or string function. An FC error may also occur as the result of: (1) A negative or exceedingly large subscript. (2) A negative or zero value for the variable in LOG. (3) A negative value for the variable in SQR. (4) A negative mantissa with non-integer exponent. (5) A call to a user-defined function for which the starting address has not been given. (6) An improper argument specified for MID$, LEFT$, RIGHT$, TAB, SPC, or ON-GOTO.
OV	**Overflow** The result of a calculation is too large for available memory. If underflow occurs, the result is zero and execution continues without error.
OM	**Out of memory** A program is too large, has too many FOR loops of GOSUB commands, too many variables, or includes expressions that are too complicated.
UL	**Undefined line** A GOTO, GOSUB, or IF-THEN-ELSE statement refers to a non-existent line number.
BS	**Subscript out of range** An array element is given either a subscript that is outside the dimensions of the array or the wrong number of subscripts.

DD	**Redimension array** Two DIM statements have been given for the same array, or a DIM statement is given only after the default dimension of 10 has been established for that array.
/O	**Division by zero** An attempt to divide by zero (or, in involution, to raise zero to a negative power) has occurred. The largest available number with the sign of the numerator is supplied as the result of the division (the largest available positive number for the result of the involution) and execution continues.
ID	**Illegal direct** A statement that is illegal in direct mode is entered as a direct mode command.
TM	**Type mismatch** A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.
OS	**Out of string space** Not enough memory free for string variables. BASIC will allocate string space dynamically until it runs out of memory.
LS	**String too long** An attempt has been made to create a string more than 255 characters long.
ST	**String formula too complex** A string expression is too long or too complex. The expression should be broken into smaller expressions.
CN	**Cannot continue** An attempt has been made to continue a program that 1) Has halted due to error 2) Has been modified during a break or 3) Does not exist.
UF	**Undefined user function.** A user defined function has been called before its definition (DEF statement) is given.

Appendix 3

Table of Characters and Codes



*This table gives the codes for all the characters on the controller keyboard. The codes are given in hexadecimal (16-based numbers). When you want to use these codes with in CHR$ to produce a character, convert them to decimal codes the following way:

Ex:The exclamation point "!". Its hexadecimal code is 21. The "2" represents its first four bits (the column number). The "1" represents its second four bits (the row number). The colum number multiplied by sixteen and added to the row number gives you the (2 × 16) + 1 = 33, decimal code for "!". So if you want to print or display an "!", use CHR$ (33).

Note:Since hexadecimal needs more than the 10 numerals we ordinarily use in the decimal system, the table uses the letters A to F to stand for the numbers 11 to 15.

A = 10 B = 11 C = 12 D = 13 E = 14 F = 15

Thus, the minus sign −, which is 80 in hexadecimal, converts to (11 × 16) + 0 = 172 in decimal.

MEMO

HATTORI SEIKO CO., LTD.

TOKYO, JAPAN

Printed in Japan
K 6F 1 K (UC BAM) 4A 15T ECS①