SD-8516 User's Guide

CHAPTER 1: How to Use the VC-3

To start using your new VC-3 system you will need to know how to load and run programs, and where to get help. This guide will be your help, and you can always find me on Discord or ask on our subreddit (r/sd8516) if you have any further questions.

Load, Save and Run programs

1. Using the LOAD and SAVE commands, you can load and save BASIC programs to the host operating system.

2. Using the DSAVE and DLOAD, you can load and save to the 1532N Super DATA-sette Tape Backup drive.

3. If you're using MON, the L and S commands use the host OS, while L, and L,FILENAME and S,FILENAME load using the 1532N. Finally, in Forth, the INCLUDE function will load from the 1532N and BROWSE command will load from the host OS.

4. If you would like to do a full tape backup of your DATA-sette Tape drive, type DTAR (data tape archive). You can restore a DTAR by typing DJOIN. This will JOIN the tape to be read in sequential order, adding any new files and replacing files that already exist.

Inserting a new Datasette is easy, just LOAD it. This is like replacing the old tape casette with a new one; you will lose your old files. If you would rather keep them and just add the files in the .TAR, you must join the tapes using the `DJOIN` (Datatape Join) command. New files from the tape archive will be added and any old files replaced.

Be sure to back up your current system configuration before you LOAD or JOIN a Datasette Archive!

Files

- You can use the DDIR command to list files.
- You can use the DDEL command to delete files.
- You can use the DLOAD and DSAVE commands to load and save from datasette,
- You can use the LOAD and SAVE commands to load and save from the host OS.

System Configuration

In order to configure your SD-8516 / VC-3 system, you will need to create a .SYSCONF file. Here's how to do it. On a fresh boot, type MON and then enter the following command:

```
00.1F
```

This will show you the 32 bytes of memory at address $0000. Now we can configure the .SYSCONF file. First, if you already have a .SYSCONF file type `0L,.SYSCONF` and it will show up here. Otherwise type the address you want to load it at, such as `C000L,.SYSCONF` and it will go to there.

If the file does not exist, you will see an error message. Then let's just make the file ourself. Here's how to do it. In MON, type:

00: CF 01: MODE (1, 2 or 6) 02: FG (0 to F) 03: BG (0 to F) 04: Colormode (1-6) 05: Boot Mode. 1: BASIC, 2: HEXMON, and 4: SD/FORTH. 06: Virtual CAPS-LOCK status. 0 = normal, 1 = CAPS-LOCK.

For example, here is the default boot-to-BASIC. If there is no .SYSCONF this will be used:

```
0: CF 1 2 9 0 0 1
```

When you are done, again type the following to verify your changes:

```
00.1F   (or just 0.F)
```

When you are ready, type:

```
0.FS,.SYSCONF
```

This will save your file to .SYSCONF. The next time you boot up, it will check .SYSCONF.

```
The sysconf32 bytes binary:
Byte 0:     $CF        ; magic (validates file)
Byte 1:     Video mode (1/2/3/6)
Byte 2:     BG color (0-15)
Byte 3:     FG color (0-15)
Byte 4:     Color mode
Byte 5:     Boot Mode
Byte 6:     VCL Mode (CAPS-LOCK)
Byte 7-31:  Reserved ($00)
```

make the SD-8516 boot into MON, try:

```
00: CF 1 2 9 0 1 1
```

To turn the SD-8516 into a 70s Forth machine, try this:

```
00: CF 6 0 8 6 4 1
```

CHAPTER 2: How to Write Programs

Writing programs for the VC-3 is Fun, Easy and Educational! There are three kinds of programming languages you can use on the SD-8516 VC-3 System:

- BASIC
- Forth

- Assembly Language

1. BASIC

You can enter BASIC programs! It's easy.

```
10 PRINT "HELLO WORLD!"
RUN
```

You can LOAD and SAVE basic programs to the host OS, or DLOAD and DSAVE them to the datasette. You can use your favorite host-os text editor to edit them, or you can use ed (see ed for more info on ed)!

If you'd like to learn more about Stellar BASIC 1.0, please see: SD-8516 Stellar BASIC.

2. FORTH

You can enter Forth programs. Just type “FORTH” to get started!

We provide similar to the Forth-79 standard plus some extra words like NUMBER and WORD. See Star Forth for more information.

3. Assembly Language

There are three ways to enter Assembly Language programs on the SD-8516.

a) HEXMON

You can enter program listings using HEXMON, or hand-assemble them yourself. For more information please see HEXMON.

b) ASSEMBLE

You can enter assembly language programs using BASIC line numbers. For more information on this technique please see SD-8516 Assembly Language.

c) ed and as

Using the ed program you can enter assembly language programs directly into the system. Then, using the as command you can assemble directly to a file. This file can be LOADed, DLOADed, or L,filename'd into MON. However, you must SYS to the correct addres or use MON to R)un it.

There is a possible fourth way. If you have an external assembler or compiler you can use that to

LOAD programs from the host OS. You can then use MON to save them if you wish.

CHAPTER 3: BEGINNING STELLAR BASIC PROGRAMMING

CHAPTER 3: BEGINNING STELLAR BASIC PROGRAMMING

Welcome to Stellar BASIC on the SD-8516! This is where the real fun begins. Stellar BASIC lets you talk directly to your machine and make it do exactly what you want. The thrill of typing in your first program and hitting RUN – the magic of watching the the screen come alive – using PETSCII graphics to play golf – the mystery of YOHO – and so much more – it's all right here, for you to discover!

Stellar BASIC is a lean, fast descendant of the famous Tiny BASIC from 1975 (the one that fit in under 4K and inspired so many early microcomputers). It's designed as a powerful yet easy to learn programming language for the everyman. There are no fancy tricks, just a straightforward list of commands. It is perfect for games, math, and little adventures, yet powerful enough to help you do taxes!

PRINT

Getting Started – The PRINT Command

The easiest way to make your SD-8516 say hello is with **PRINT**. You can also use the ? key as shorthand.

Type this right now (press ENTER after the line):

```
? "HELLO WORLD"
```

The computer should immediately display:

```
  HELLO WORLD
```

Now try:

```
? 42 + 8
```

It prints 50. You can do math right away, no program needed. This is called **direct mode,** directly entering commands on the terminal.

GOTO

Your First Program – The Never-Ending Message

Let's write a real program. Type **NEW** and press ENTER to clear any old stuff.

Now carefully type these lines (press ENTER after each one):

```
10 PRINT "HELLO WORLD"
20 GOTO 10
```

Type **RUN** and press ENTER.

Watch the screen fill up with “HELLO WORLD” scrolling forever! To stop the program, press the **BREAK** or **STOP** key (ESC on modern keyboards).

PROGRAMS

Line Numbers and How Programs Work

Every line in a program starts with a **line number** (any whole number from 1 to around 65000). The computer runs lines in order of those numbers, smallest to largest.

You don't have to type lines in order, the computer will sort them for you automatically. Try this:

```
20 ? "LINE 20 COMES FIRST ANYWAY!"
10 ? "BUT LINE 10 RUNS FIRST"
```

RUN

LIST

To see your program again, type **LIST**.

To erase just one line, type its number and press ENTER (e.g., 20 [ENTER] deletes line 20).

If you type **NEW** it will delete the current program. Make sure you SAVE “NAME” your program first.

LOAD and SAVE

As you learn more about BASIC programming and your programs get longer, you will begin to want a way to save them; Also, to share them and easily load them! The two commands you should learn next are LOAD and SAVE. LOAD will load a program from the disk and SAVE will save the program in memory to disk. Also try “SAVE <FILENAME> to save your programs! And, even if you just type “SAVE” on a line by itself the computer will choose a progam name for you.

LET (VARIABLES)

Variables – Remembering Things

Variables are like little memory boxes. In Stellar BASIC we use single letters only: **A**, **B**, **Z**, etc. You can also use letters with a $ for strings (ex A$). More on strings soon. For now let's deal with single letters, which are used for numbers.

Try this in direct mode:

```
LET A = 25

? A
```

It prints 25.

Now a tiny counting program:

```
10 LET A = 1
20 ? A
30 LET A = A + 1
40 IF A <= 10 THEN GOTO 20
50 ? "BLAST OFF!"
```

RUN

This program counts from 1 to 10. Can you make it count down from 10 to 1 instead?

DIM (ARRAYS)

You can use the DIM(number) command to pre-allocate space for arrays; however strictly speaking it is not necessary to do so, any uninitialized array variable will return zero.

Here's a short example of the DIM command. You can also DIM a string varible; A$(3) is a string.

```
    10 REM ARRAY TEST PROGRAM
    20 DIM A(5)
    30 REM SET VALUES OUT OF ORDER
    40 LET A(3) = 99
    50 LET A(1) = 42
    60 LET A(5) = 77
    70 LET A(0) = 10
    80 LET A(4) = 55
    90 LET A(2) = 31
    100 REM READ BACK IN SEQUENTIAL ORDER
    110 FOR I = 0 TO 6
    120 PRINT "A(";I;") = ";A(I)
    130 NEXT I
    140 REM EXPECTED:
    150 REM A(0) = 10
```

```
    160 REM A(1) = 42
    170 REM A(2) = 31
    180 REM A(3) = 99
    190 REM A(4) = 55
    200 REM A(5) = 77
    210 PRINT
    220 REM OVERWRITE AND RE-READ
    230 LET A(3) = 0
    240 LET A(1) = 999
    250 PRINT "A(1) = ";A(1);" EXPECT 999"
    260 PRINT "A(3) = ";A(3);" EXPECT 0"
    270 PRINT "A(5) = ";A(5);" EXPECT 77"
    280 PRINT
    290 REM TEST OUT OF BOUNDS
    300 PRINT "TESTING A(6)..."
    310 LET A(6) = 123
    320 PRINT "A(6) = ";A(6)
    330 PRINT
    340 PRINT "DONE"
```

The DIM statement will cause undefined variables to BREAK your program in the future, so it is suggested that you use it. It will also make clear to people how many of each kind of variable to expect. It also has the immediate practical use of allocating space for the variables in memory, which means it makes sure there is enough memory to hold your variable without having to crash in the middle of the program!

INPUT

The INPUT Command – Asking Questions

Now let's make the computer ask questions.

Type **NEW**, then:

```
10 PRINT "WHAT IS YOUR GUESS (1-100)";
20 INPUT E
30 PRINT "YOU ENTERED: "; E
```

RUN

Type a number and press ENTER. Notice the semicolon (;) keeps things on the same line–no extra ? prompt needed.

GET

INPUT's best friends is GET. GET is non-blocking, while INPUT is blocking. The following program

shows you how this works:

```
    10 GET A$
    20 IF A$ = "" THEN GOTO 10
    30 PRINT "OK! "; A$
```

As you can see, the program waits for you to press a key, then exits.

IF-THEN

Making Decisions with IF…THEN

```
  "The problem is choice." -Neo
```

The real power comes from decisions. Once your programs can do different things based on their environment they become useful and powerful.

Add these lines to the above program make a simple answer checker:

```
40 IF E = 42 THEN GOTO 70
50 IF E <> 42 THEN ? "TRY AGAIN!"
60 GOTO 10
70 PRINT "YOU FOUND THE ANSWER!"
```

RUN and try a guess until you hit 42. (42 is a reference to the Hitchhiker's Guide – a great book).

Your First Game

A Classic: Number Guessing Game

```
  WOULD YOU LIKE TO PLAY A GAME (Y/N)?
```

Here's a complete small game you can type in; it's the first program ever written in Stellar BASIC.

It uses everything we've covered: PRINT/?, variables, INPUT, IF-THEN, GOTO, and introducing a new function: RAND() for random numbers.

Type **NEW** to start a new program, then enter these codes into your computer:

```
10 ? ""
20 ? "NUMBER GUESSING GAME"
30 ? "GUESS MY NUMBER 1-100!"
40 LET A = 1
50 LET B = 100
```

```
60 LET D = RAND( B - A + 1 ) + A - 1   : REM SECRET NUMBER!
70 LET F = 0
80 LET F = F + 1
90 PRINT ""
100 PRINT "ROUND";F
110 PRINT "LOW:";A;" HIGH:";B
120 INPUT "YOUR GUESS:"; E
130 IF E > B THEN PRINT "TOO LARGE!" : GOTO 90
140 IF E < A THEN PRINT "TOO SMALL!" : GOTO 90
150 IF E < D THEN PRINT "HIGHER!" : LET A = E : GOTO 80
160 IF E > D THEN PRINT "LOWER!" : LET B = E : GOTO 80
170 PRINT "YOU GOT IT!"
180 PRINT "SCORE:";101-F
190 PRINT "PLAY AGAIN? ( 1=Y / 0=N ) ";
200 INPUT G
210 IF G = 1 THEN GOTO 50
220 PRINT "BYE FOR NOW!"
```

RUN and play! (Note: RAND(N) gives 0 to N-1, so we adjust +A-1 to get the range.)

This is the spirit of 8-bit BASIC–type it in, play it, tweak it, make it yours. Just like the old days when you'd stay up late copying listings from magazines.

IF-THEN EXAMPLE

Here is an example program:

```
10 LET A = 5
20 PRINT A
30 LET A = A - 1
40 IF A > 0 THEN GOTO 20
50 PRINT "GOAL!"
```

Next, type **RUN.** You will see a countdowns to 1, then “GOAL!”

GOSUB and RETURN

A useful feature of BASIC is **GOSUB** and **RETURN**. Think of a subroutine as a little helper routine you can call from anywhere. GOSUB jumps to it (and remembers where you came from), and RETURN sends you right back to the next line.

This saves tons of typing–perfect when you want the same message or calculation in multiple spots.

Here is an example of a GOSUB helper function:

```
10 REM GOSUB SQUARE DEMO
```

```
20 LET A = 2
30 GOSUB 1000
40 LET A = 3
50 GOSUB 1000
60 LET A = 4
70 GOSUB 1000
80 LET A = 5
90 GOSUB 1000
100 GOTO 9000
9000 REM END

1000 REM THIS HELPER FUNCTION WILL PRINT THE SQUARE OF A
1010 PRINT A * A
1020 RETURN
```

After you enter this program and type RUN, you will see the result 4, 9, 16, 25. As you can see, every time the program calls **GOSUB 1000,** it runs the code at LINE NO. 1000 and then **RETURN**s to continue in the main program.

You can nest up to 8 **GOSUB** calls in one program.

Improved Number Guessing Game with GOSUB

Let's level up the guessing game. Let's use GOSUB for separate “display instructions” and “show result” routines. Cleaner code, easier to tweak.

NEW then type:

```
10 PRINT ""
20 GOSUB 900   : REM SHOW INSTRUCTIONS
30 A=1 : B=100
40 D=RAND(B-A+1)+A   : REM SECRET NUMBER (1-100)
50 F=0

60 F=F+1
70 ? ""
80 ? "ROUND";F
90 ? "LOW:";A;" HIGH:";B
100 INPUT "YOUR GUESS:"; E
110 IF E>B THEN GOSUB 800:GOTO 70
120 IF E<A THEN GOSUB 700:GOTO 70
130 IF E<D THEN GOSUB 600:A=E:GOTO 60
140 IF E>D THEN GOSUB 500:B=E:GOTO 60

150 GOSUB 400
160 ? "SCORE:";101-F
170 ? "PLAY AGAIN? (Y=1/N-0)"
180 INPUT G
190 IF G = 1 THEN GOTO 30
200 ? "THANKS FOR PLAYING!"
```

```
210 GOTO 5000

400 ? "YOU GOT IT IN";F;" GUESSES!"
410 RETURN

500 ? "LOWER!"
510 RETURN

600 ? "HIGHER!"
610 RETURN

700 ? "TOO SMALL!"
710 RETURN

800 ? "TOO LARGE!"
810 RETURN

900 ? "NUMBER GUESSING GAME V1.1"
910 ? "BY APPLEDOG (C) 2026"
920 ? "GUESS THE NUMBER BETWEEN 1 AND 100."
930 ? "I'LL TELL YOU HIGHER OR LOWER."
940 RETURN
5000 REM END
```

Try to **RUN** this program and play a few rounds! The subroutines make it modular. If you want fancier win/lose messages later, just edit those GOSUB lines.

Negative Numbers in Action

Since negatives are fully supported, try tweaking the game or make a countdown timer:

```
10 PRINT "COUNTDOWN FROM ZERO..."
20 LET A = 0
30 PRINT A
40 LET A = A - 1
50 IF A >= - 10 THEN GOTO 30
60 PRINT "BLAST OFF INTO NEGATIVE SPACE!"
```

Strings

What's a String? A string is a word or a sentence. It's not a number. This is a number variable:

```
    10 LET A = 2
    20 PRINT A
```

And this is a string:

```
10 LET A$ = "HELLO WORLD!"
20 PRINT A$
```

They call it a string because it is a “string of bytes”. A string of *letters* in this case.

You can do all sorts of things with strings. There are also functions that can help you with strings, like LEN, ASC and VAL.

```
10 LET A$ = "HELLO WORLD"
20 PRINT "=== LEN ==="
30 PRINT LEN(A$)
40 PRINT "=== ASC ==="
50 PRINT ASC(A$)
60 PRINT ASC("Z")
70 PRINT "=== VAL ==="
80 LET B$ = "42"
90 PRINT VAL(B$)
100 LET X = VAL(B$) + 8
110 PRINT X
```

The expected output is:

```
=== LEN ===
11
=== ASC ===
72
90
=== VAL ===
42
50
```

This can have all sorts of uses. With a little math you can interpret hexidecimal numbers, or write subroutines that encode and decode secret messages! As you see, you can also store numbers in strings. You can use this to have more than the default number of integer variables. With a little thought, you could even use this to add two very large numbers together. The sky's the limit!

More Strings!

We give you the mighty CHR$, LEFT$, RIGHT$ and MID$. They are your *secret weapons* in the *string war.* Don't worry, there's no string war. I just made that up right now because it sounded cool.

```
10 LET A$ = "HELLO WORLD"
20 PRINT "=== CHR$ ==="
30 LET C$ = CHR$(65)
40 PRINT C$
50 LET C$ = CHR$(72) + CHR$(73)
```

```
60 PRINT C$
70 PRINT "=== LEFT$ ==="
80 PRINT LEFT$(A$,5)
90 PRINT "=== RIGHT$ ==="
100 PRINT RIGHT$(A$,3)
110 PRINT "=== MID$ ==="
120 PRINT MID$(A$,7,4)
130 PRINT "=== STR$ ==="
140 LET D$ = STR$(123)
150 PRINT D$
160 LET D$ = "VALUE=" + STR$(99)
170 PRINT D$
```

The expected output is:

```
=== CHR$ ===
A
HI
=== LEFT$ ===
HELLO
=== RIGHT$ ===
RLD
=== MID$ ===
WORL
=== STR$ ===
123
VALUE=99
```

There are many things you can do with strings.

Screen Commands

CLS: A Fun Command

Type CLS at the READY prompt. What happens? You can use this command to clear the screen. This is great if you want to create an impression for your programs. Start fresh, clear away any distractions.

COLORMODE

You may wish to change to one of the other default palettes. For example, for a classic C64 look, try this:

```
10 FG 14
20 BG 6
30 COLORMODE 3
```

Or if you're into the old-school VIC-20, try

```
  COLORMODE 1 : BG 1 : FG 14
```

There are a large variety of colors and modes you can try! Here are some suggested values:

Year	System	COLORMODE	FG	BG
1975	DEC V52	5 (5153)	11	8
1977	APPLE][	0 (RETRO)	9	5
1980	VIC-20	2 (1702)	14	1
1981	PC	4 (CGA)	7	0
1981	BBC MICRO	0 (RETRO)	3	1
1982	C64	3 (1084)	14	6
1983	DEC VT200	0 (RETRO)	15	12
1983	DEC VT220	0 (RETRO)	8	4

PALETTE

Some color combinations are hard to match! For example due to the phosphor and the glass on a terminal like the WANG 2200 or the Datapoint 3300, the default palettes are not able to represent the colors accurately. For situations like this you will want to use custom color values. Here's what you need to do:

```
    10 REM WANG 2200
    20 LET P = 10
    30 LET I = 10 * 16
    40 COLORMODE P
    50 FG 8
    60 BG 1
    70 PALETTE I,0,20,10
    80 PALETTE I+1,0,26,13
    90 PALETTE I+2,0,32,16
    100 PALETTE I+3,0,40,24
    110 PALETTE I+4,28,15,0
    120 PALETTE I+5,74,255,0
    130 PALETTE I+8,0,255,153
    140 PALETTE I+9,0,230,140
    150 PALETTE I+10,0,255,180
    160 PALETTE I+11,64,255,192
    170 PALETTE I+12,255,183,0
    180 PALETTE I+13,18,15,0
```

With the above color codes set, you can now do this:

Year	System	COLORMODE	FG	BG
1969	DATAPOINT 3300	10 (USER)	10	3
1973	WANG 2200	10 (USER)	9	1
1971	IBM 3270 “Greenscreen”	10 (USER)	13	5

Year	System	COLORMODE	FG	BG
1986	WYSE WY-60	10 (USER)	12	4

More Loops

Stellar BASIC V1 keeps things simple and fast; Originally there was not even a FOR-NEXT loop, and even now there is no WHILE or DO loop. But you can still create powerful repeating loops using just IF…THEN and GOTO. This section will demonstrate the kind of clever thinking you will need to write advanced programs in TinyBASIC.

The key tricks:

- WHILE – A while loop checks the condition first, so place the IF check before the loop body.
- DO-WHILE – A do loop runs at least once, so place the IF check at the end of the loop body.
- FOR-NEXT – A for-next is essentially a WHILE loop (see below).

Let's see these in action with short examples you can type in right now.

WHILE

WHILE is short for WHILE-DO. The loop check is at the front of the do-loop (so it may skip the loop entirely). This is like “while something is true, keep doing the body.”

Example: Print numbers from 1 to 5, checking first.

```
10 LET I = 1
20 IF I > 5 THEN GOTO 60
30 ? I
40 LET I = I + 1
50 GOTO 20
60 ? "DONE!"
```

This prints 1 through 5. If you change line 10 to LET I=10, it skips the printing entirely, since the condition was false when we checked the loop condition (at the front of the loop body).

DO-WHILE

In this style the do-loop is executed and then if the condition (the while) passes, it executes the loop again. This style of loop always runs at least once.

Example: Keep asking for a positive number until you get one.

```
10 INPUT "ENTER A POSITIVE NUMBER ", N
20 IF N > 0 THEN GOTO 40
30 PRINT "TRY AGAIN -- MUST BE POSITIVE!"
40 IF N <= 0 THEN GOTO 10
```

```
50 PRINT "THANKS! YOU ENTERED "; N
```

This program runs the INPUT at least once. If you enter -5, it complains and loops back. If positive, it exits. The check is at the bottom; no skip on first pass.

Simulated FOR-NEXT

Although BASIC typically includes a FOR-NEXT construct, you do not need to use it (although it is simpler than writing out the loop logic using IF and GOTO.)

The idea behind FOR-NEXT is to iterate over a range of values, and the NEXT check typically occurs at the front of the loop. However, knowing how to simulate your own loops adds flexibility and can make some kinds of looping possible that you just can't do with FOR-NEXT.

Example #1: Countdown from 10 to 0, check at top.

```
10 LET C = 10
20 IF C >= 0 THEN GOTO 40
30 GOTO 70
40 PRINT C
50 LET C = C - 1
60 GOTO 20
70 PRINT "IGNITION!"
```

Observe that sometimes flipping the logic of the test makes the code cleaner (no extra check):

```
10 LET C = 10
20 IF C < 0 THEN GOTO 60
40 PRINT C
50 LET C = C - 1
60 GOTO 20
70 PRINT "IGNITION!"
```

In this second case, inverting the logic falls-through to the do-loop. This is more efficient, even though the idea is “if C is greater than or equal to zero”. Know your logic operators!

Here is another example of a similar loop, that amounts to a FOR-NEXT loop.

Example #2: Print even numbers 2 to 20 (step +2).

```
10 LET X = 2
20 IF X > 20 THEN GOTO 60
30 PRINT X
40 LET X = X + 2
50 GOTO 20
```

```
60 PRINT "EVEN NUMBERS DONE!"
```

Or, to simulate STEP -1 (countdown), as before,

```
10 LET Y = 20
20 IF Y < 1 THEN GOTO 60
30 PRINT Y
40 LET Y = Y - 1
50 GOTO 20
60 PRINT "BLAST OFF!"
```

These methods of program flow control are from old-school TinyBASIC programming. This was the norm – inventive uses of code. There is a LOT you can do with TinyBASIC. It's understandable, and very efficient.

Further Study

You can practice your BASIC skills with these exercises:

- Make the WHILE example count backwards.
- Turn the guessing game's round counter into a DO-style loop (run at least one round?).
- Add a “Play again?” wrapper around your game.

When FOR-NEXT arrives in a future update, you'll appreciate how these hand-built loops taught you control flow. Until then, don't forget to “think differently!”

PEEK

The PEEK command allows you to read memory from the computer. There are many things you can do with this command. Note that the BANK command sets which bank (0, 1, 2 or 3) you will PEEK from. Here is a short example of some things you can do with the PEEK command:

BANK	LOCATION	DESCRIPTION
BANK 1	PEEK(61184)	Get current video mode.
BANK 1	PEEK(61185)	Get max columns of current video mode.
BANK 1	PEEK(61189)	Video clock low byte (four bytes in length).
BANK 1	PEEK(61198)	Cursor column.
BANK 1	PEEK(61199)	Cursor row.
BANK 1	PEEK(61201)	Number of keys in keyboard buffer.
BANK 1	PEEK(61202)	First byte pair of keyboard buffer.
BANK 1	PEEK(61251)	Palette mode.
BANK 1	PEEK(61440)	TEXT 0,0 in mode 1
BANK 1	PEEK(63488)	COLOR 0,0 in mode 1
BANK 1	PEEK(61440)	TEXT 0,0 in mode 2
BANK 1	PEEK(63488)	COLOR 0,0 in mode 2

Of course, if you want to use the zero page for extra memory, you can switch to BANK 0 and then PEEK (0) to PEEK(255) to read values from there, too. Have fun and explore everything you can read with PEEK!

POKE

The POKE command works with BANK to write to a memory location. For example, to add a key to the keyboard buffer, you can POKE 61202,65 then POKE 61201,1 to indicate there is a key in the buffer. For example, try the following fun program:

```
10 POKE 61202,82
20 POKE 61204,85
30 POKE 61206,78
40 POKE 61208,13
50 POKE 61201,4
```

The numbers 82, 85, 78 and 13 represent **“RUN<ENTER>”**.

If you RUN this program, it will add **“RUN<ENTER>”** to the keyboard buffer, causing the program to run again (!!) You can break out of this endless loop by pressing the BREAK key (ESC on modern keyboards).

READ, DATA, RESTORE

If you have large amounts of data that you need to store for your program, you can use DATA.

```
10 DATA 1, 3, 5, 7, 9, 10, 11, 12, 13, 15
20 FOR I = 1 TO 10
30 READ A
40 PRINT I, A
50 NEXT I
```

You can also have DATA at the end of the program; this works too:

```
20 FOR I = 1 TO 10
30 READ A
40 PRINT I, A
50 NEXT I
100 DATA 2, 3, 4, 5, 10, 20, 30, 50, 70, 99
```

NOTE: DATA is currently BROKEN. It only uses the first line of DATA. I will fix this for the May 4th 2026 update.

Examples of DATA you can store could be:

- Pixels for an image or a sprite
- Monster data

- Maps for an adventure game
- Musical notes
- Secret data (i.e. ASCII encoded text, so people don't cheat by LISTing your program!)

MATH functions

There are some additional functions you may need while programming more advanced applications. The ABS, MIN, MAX, SGN and SQRT functions will help you perform mathematics in BASIC. Here's a short demo program illustrating how these math functions are used.

```
10 REM === ABS TEST ===
20 PRINT "ABS(-5)="; ABS(-5)
30 PRINT "ABS(5)="; ABS(5)
40 PRINT "ABS(0)="; ABS(0)
50 PRINT ""
60 REM === SGN TEST ===
70 PRINT "SGN(-99)="; SGN(-99)
80 PRINT "SGN(0)="; SGN(0)
90 PRINT "SGN(42)="; SGN(42)
100 PRINT ""
110 REM === MIN/MAX TEST ===
120 PRINT "MIN(3,7)="; MIN(3,7)
130 PRINT "MIN(7,3)="; MIN(7,3)
140 PRINT "MAX(3,7)="; MAX(3,7)
150 PRINT "MAX(7,3)="; MAX(7,3)
160 PRINT "MIN(-5,5)="; MIN(-5,5)
170 PRINT "MAX(-5,5)="; MAX(-5,5)
180 PRINT ""
190 REM === SQR TEST ===
200 PRINT "SQR(0)="; SQR(0)
210 PRINT "SQR(1)="; SQR(1)
220 PRINT "SQR(4)="; SQR(4)
230 PRINT "SQR(9)="; SQR(9)
240 PRINT "SQR(16)="; SQR(16)
250 PRINT "SQR(100)="; SQR(100)
260 PRINT "SQR(144)="; SQR(144)
270 PRINT "SQR(10)="; SQR(10)
280 PRINT ""
290 REM === PRACTICAL USE ===
300 FOR I = -5 TO 5
310 PRINT I; ":ABS="; ABS(I); " SGN="; SGN(I)
320 NEXT I
```

Expected output:

```
ABS(-5)=5
ABS(5)=5
ABS(0)=0
```

```
SGN(-99)=-1
SGN( 0)=0
SGN(42)=1

MIN(3,7)=3
MIN(7,3)=3
MAX(3,7)=7
MAX(7,3)=7
MIN(-5,5)=-5
MAX(-5,5)=5

SQR(0)=0
SQR(1)=1
SQR( 4)=2
SQR(9)=3
SQR(16)=4
SQR(100)=10
SQR(144)=12
SQR(10)=3

-5:ABS=5 SGN=-1
-4:ABS=4 SGN=-1
-3:ABS=3 SGN=-1
-2:ABS=2 SGN=-1
-1:ABS=1 SGN=-1
0:ABS=0 SGN=0
1:ABS=1 SGN=1
2:ABS=2 SGN=1
3:ABS=3 SGN=1
4:ABS=4 SGN=1
5:ABS=5 SGN=1
```

CHAPTER 4: How to write games in Stellar BASIC

In this chapter we will demonstrate some advanced programming techniques by writing a real game in Stellar BASIC. Today's game is ROBOT CHASE, the classic game that inspired BSDgames' 'robots'.

robots or CHASE?

'Robots' is not an original game; it is a direct descendant of an earlier game called Chase, a turn-based pursuit game that likely originated in the early 1970s on the Dartmouth Time-Sharing System (DTSS) at Dartmouth College. The author of the original Chase remains unknown, but public versions appeared in Creative Computing magazine in 1976, with various modifications circulating afterward. Chase involved escaping from pursuing robots in a simple grid environment.

Development of the BSD Version

In November 1984, Allan R. Black developed a Unix implementation of the game. He posted it to the Usenet newsgroup net.sources.games in May 1985.

Ken Arnold (a key figure in early BSD development and co-contributor to the legendary roguelike Rogue) then ported and integrated it into the Berkeley Software Distribution. The BSD version of robots first appeared in 4.3BSD in June 1986.

Ken Arnold is often credited as the primary author or maintainer in BSD man pages for the game, reflecting his role in refining and including it in the official bsdgames package.

Legacy

Today, `robots` fits perfectly into this era of 1980s Unix gaming: simple, keyboard-driven (often using vi-style movement keys like h/j/k/l or numeric keypad), curses-based for terminal rendering, and highly replayable due to procedural level generation. It has been compared to a turn-based version of arcade games like Robotron: 2084 or Berzerk, but with the strategic depth of dodging and luring robots into collisions.

The game remains popular among retro computing enthusiasts. It has been ported or reimplemented countless times. And today, we will write a version of it in BASIC for the SD-8516.

The Game Loop

One way of looking at program design is the typical “input loop”, or, “game loop”. This loop has three basic functions:

```
a) Display what the user needs to see,
b) Get input from the user on what he wants to do
c) Process the result
```

This 'loop' continues until the user chooses to quit the program.

SECTION A) DRAW THE MAP

To begin, let us look at section a, the first step, which is 'showing the user what he needs to see'. We will begin by drawing the map. You could start with the game loop first and plan out “what the user can do”, but I find that having something cool to look at first gets my creativity flowing.

```
1000 REM DRAW MAP
1010 CLS
1020 LET A = ASC("*")
1030 FOR X = 0 TO 39
1040 CHARXY X, 0, A
1050 CHARXY X, 24, A
```

```
   1060 NEXT X
   1070 FOR Y = 0 TO 24
   1080 CHARXY 0, Y, A
   1090 CHARXY 39, Y, A
   1100 NEXT Y
   1900 RETURN
```

This is a subroutine that draws the basic map. You call it like this:

```
  GOSUB 1000
```

It will draw the basic map and then return.

Let's make it more exciting; let's draw the player, and a robot.

```
   10 LET PX = 19
   20 LET PY = 11
   30 LET RX = 1
   40 LET RY = 1
```

The above is just to define the position of the player (PX,PY) and the evil, murderous robot (RX, RY). Do not let the “RX” fool you; this is not a medical robot.

Let's now draw the position of the player and the robot using the CHARXY function:

```
   1200 CHARXY PX, PY, ASC("@")
   1300 CHARXY RX, RY, ASC("r")
```

Section B) Player Input Loop

There are three kinds of input loop: INPUT, HASKEY and GETKEY. We will use a GETKEY loop for this game. First let's look at all the different types of input loop and then we will show the code for our game.

```
  10 INPUT "Enter a number from 1 to 10 (0 to quit): ", A
  20 IF A = 0 GOTO 60
  30 PRINT "You typed: ", A
  40 PRINT ""
  50 GOTO 10
  60 PRINT "END OF LINE"
```

The above program demonstrates an INPUT loop. The INPUT loop allows the user to type a line of text, which he can submit via ENTER. The program then processes the user input and returns to the top of the loop.

```
  10 PRINT "PRESS ANY KEY OR 'X' TO END: "
```

```
20 LET K = GETKEY()
30 IF K = 88 THEN END
40 PRINT CHR$(K);
50 GOTO 20
```

The above is is a GETKEY loop. It only accepts a single keystroke of input before exiting.

```
10 PRINT "PRESS ANY KEY OR 'X' TO END: "
20 IF HASKEY() = 0 THEN GOTO 20
25 LET K = GETKEY()
30 IF K = 88 THEN END
40 PRINT CHR$(K);
50 GOTO 20
```

In contrast to a GETKEY loop, a HASKEY loop waits to see if there is a key. This is important because you might want to keep the game running in the background while you wait for a key. Notice that the HASKEY loop and the GETKEY loop both work using GETKEY(), however, the haskey loop only tries to get a key when one is available, allowing it to quickly check other things before coming back to look for an input key.

```
10 PRINT "PRESS ANY KEY OR 'X' TO END: "
20 LET N = HASKEY()
30 IF N = 0 THEN GOSUB 100
40 IF N = 1 THEN GOSUB 200
50 GOTO 20
100 REM NO KEY
110 PRINT "*"
120 WAIT 200
130 RETURN
200 REM HAS KEY
210 LET K = GETKEY()
220 IF K = 88 THEN END
230 PRINT " *** " ; CHR$(K); " ***"
240 RETURN
```

The above demo program checks for keys 5 times a second (WAIT 200). To check 10 times a second, try WAIT 100 (i.e. 100ms). Instead of WAIT, the program could be doing other things, like playing music or animating sprites! That's the power of a HASKEY loop – but not every program needs to use that kind of input loop.

Our ROBOTS input loop

Let's put our input loop at 2000. This is a subroutine that we will call when we want to ask the player what his move is.

```
2000 REM *** SECTION B) INPUT LOOP
2110 LET K = GETKEY()
2120 LET L = ASC("H")
```

```
2130 LET D = ASC("J")
2140 LET U = ASC("K")
2050 LET R = ASC("L")
2060 LET Q = ASC("Q")
2070 IF K = L THEN GOSUB 5000 : REM MOVE LEFT
2080 IF K = D THEN GOSUB 5100 : REM MOVE DOWN
2090 IF K = U THEN GOSUB 5200 : REM MOVE UP
2100 IF K = R THEN GOSUB 5300 : REM MOVE RIGHT
2110 IF K = Q THEN GOSUB 5400 : REM QUIT
2120 RETURN
```

For now, this is our input loop. We are just laying out a plan here; get a key and move the player. We haven't written anything else yet, but we have an idea of where we are; this is “section b” of the a-b-c three-section game plan.

SECTION C) PROCESS DATA

The next step is to move the player; i.e. deal with the player's input.

```
5000 REM MOVE PLAYER LEFT
5010 LET PX = PX - 1
5020 IF PX < 1 THEN LET PX = 1
5030 RETURN
```

Here, we check to see if PX is 0 or less (i.e. on the border). If this is true, we do not allow the player to move onto the edge of the game board. The other cases are similar:

```
5000 REM MOVE PLAYER LEFT
5010 LET PX = PX - 1
5020 IF PX < 1 THEN LET PX = 1
5030 RETURN
5100 REM MOVE PLAYER DOWN
5110 LET PY = PY + 1
5120 IF PY > 22 THEN LET PY = 22
5130 RETURN
5200 REM MOVE PLAYER UP
5210 LET PY = PY - 1
5220 IF PY < 1 THEN LET PY = 1
5230 RETURN
5300 REM MOVE PLAYER RIGHT
5310 LET PX = PX + 1
5320 IF PX > 38 THEN PX = 38
5330 RETURN
```

Phase I Testing

Here's the complete listing so far. I have added some code at line no. 100 to 200 to run the game loop. You can press Q to quit. Try it!

```
10 LET PX = 19
20 LET PY = 11
30 LET RX = 1
40 LET RY = 1
100 REM DRAW MAP
110 GOSUB 1000
120 REM GET PLAYER INPUT
130 GOSUB 2000
140 REM CHECK COLLISION
150 IF PX = RX THEN IF PY = RY THEN GOTO 4000
180 REM LOOP UNTIL QUIT
190 GOTO 100
1000 REM DRAW MAP
1010 CLS
1020 LET A = ASC("*")
1030 FOR X = 0 TO 39
1040 CHARXY X, 0, A
1050 CHARXY X, 24, A
1060 NEXT X
1070 FOR Y = 0 TO 24
1080 CHARXY 0, Y, A
1090 CHARXY 39, Y, A
1100 NEXT Y
1110 LET PC = ASC("@")
1120 LET RC = ASC("r")
1200 CHARXY PX, PY, PC
1300 CHARXY RX, RY, RC
1900 RETURN
2000 REM *** SECTION B) INPUT LOOP
2010 LET K = GETKEY()
2020 LET L = ASC("H")
2030 LET D = ASC("J")
2040 LET U = ASC("K")
2050 LET R = ASC("L")
2060 LET Q = ASC("Q")
2200 IF K = L THEN GOSUB 5000 : REM MOVE LEFT
2210 IF K = D THEN GOSUB 5100 : REM MOVE DOWN
2220 IF K = U THEN GOSUB 5200 : REM MOVE UP
2230 IF K = R THEN GOSUB 5300 : REM MOVE RIGHT
2240 IF K = Q THEN GOSUB 5400 : REM QUIT
2900 RETURN
5000 REM MOVE PLAYER LEFT
5010 LET PX = PX - 1
5020 IF PX < 1 THEN LET PX = 1
```

```
5030 RETURN
5100 REM MOVE PLAYER DOWN
5110 LET PY = PY + 1
5120 IF PY > 22 THEN LET PY = 22
5130 RETURN
5200 REM MOVE PLAYER UP
5210 LET PY = PY - 1
5220 IF PY < 1 THEN LET PY = 1
5230 RETURN
5300 REM MOVE PLAYER RIGHT
5310 LET PX = PX + 1
5320 IF PX > 38 THEN LET PX = 38
5330 RETURN
5400 REM QUIT
5410 CLS
5420 PRINT "QUIT GAME"
5430 PRINT "THANK YOU FOR PLAYING!"
5440 END
```

This program is great so far but it can be improved.

Finishing Touches

The first thing is, the screen flashes when you're playing. That can be solved using VSTOP and VSTART to pause screen updates. Try adding these lines:

```
1005 VSTOP
1800 VSTART
```

Second, if you've entered this program using the Autotyper, or if INVERT CAPS is on *(“CAPS” at the Ready prompt), the program will not recognize lowercase letters. The simple fix is to convert lowercase to uppercase:

```
2015 IF K >= 97 THEN LET K = K - 32
```

Third, if you walk on top of the robot (or, if the robot walks on top of you) nothing happens. You're supposed to avoid the robot – the robot is your rival.

```
2500 IF PX = RX THEN IF PY = RY THEN GOTO 4000
4000 REM ROBOT CATCHES YOU
4010 CLS
4020 ? "OH NO! THE ROBOT CATCHES YOU."
4030 ? "GAME OVER."
4900 END
```

This would be a boring game if the robot didn't try and catch you! Here's how to make the robot move:

```
140 REM MAKE ROBOTS MOVE
150 GOSUB 7000
7000 REM MAKE ROBOT MOVE TOWARDS PLAYER
7010 IF RX < PX THEN LET RX = RX + 1
7020 IF RX > PX THEN LET RX = RX - 1
7030 IF RY < PY THEN LET RY = RY + 1
7040 IF RY > PY THEN LET RY = RY - 1
7900 RETURN
```

And let's also add a beep every time the robots move, to add to their dread. You can read more about the PLAY command in Music.

```
7100 PLAY "T120 W1 V5 O2 L24 B"
```

If you play the game, you will notice the collision check doesn't work. Why? It's because you are checking for a collision on line 2500 before you move the robots. You need to check after. Delete line 2500 and type it in again as line 160:

```
160 REM CHECK FOR COLLISION
170 IF PX = RX THEN IF PY = RY THEN GOTO 4000
```

After this, you have the basis for a playable game. You could add:

- Levels with more robots – using arrays, for example RX(3) for the third robot's RX.
- A stone (junk pile) that kills robots when they touch it (they crash).
- Robot collisions which create crash-piles
- Scores, including a hi-score list
- A teleport function (max three times per level!)
- Anything you can think of!

ROBOTS.BAS Complete Program

The source code for ROBOTS.BAS is listed on the BASIC Programs page:

- Stellar BASIC Programs
- Direct link: ROBOTS.BAS source code

NEXT STEPS

What's Next?

You've now got the core toolkit: PRINT, LET variables, INPUT, IF-THEN, GOTO, RAND(), and GOSUB/RETURN for structured programs. Experiment wildly, change freely– add more subroutines, make new games, introduce more features!

You can also check out some programs that users just like you have submitted to our Stellar BASIC Programs archive!

You are in the driver's seat now! Worlds of adventure await your discovery!

*Don't forget to check out the **SD-8516 Programmer's Reference Guide** and **SD-8516 Stellar BASIC** for more advanced tutorials dealing with character, graphics and sound commands!*