Saving Memory on the Sinclair ZX81

Sean A. Irvine
Hamilton, New Zealand
sairvin@gmail.com

In 1981, Sinclair Research released the ZX81 microcomputer, or Timex 1000 as it was known in the United States. The ZX81 was based around the 8-bit Z80 microprocessor running at 3.25 MHz with an 8 K read-only memory containing the operating system and a BASIC language interpreter. In the smallest configuration the ZX81 had just 1 K of RAM. An extra RAM pack this could be increase this to 16 K, and some third-party packs supported larger memory sizes. Whatever RAM was available, needed to be shared between the display, any program the user may have loaded or typed in, and any state associated with the current execution. Despite the low amount of memory some remarkable feats were achieved, including an implementation of chess that ran on the 1 K version [17, 18, 19, 20].

In 1982, Sinclair Research released the ZX Spectrum which had 48 K in the most common configuration. The focus here is on the ZX81, but the techniques discussed are applicable to the ZX Spectrum; although with 48 K its memory problems were less acute. The ZX Spectrum had the first ever official implementation of the Scrabble® board game, supporting a vocabulary of over 11 000 words in 48 K [30], comparable to the Unix `spell` command of that era [29].

We concentrate on the representation of numbers in BASIC, but in practice tricks for representing numbers were combined with other memory saving strategies. Many of these techniques first appeared in the popular magazines associated with personal computing and read like the antithesis of modern programming: avoid comments, use the shortest variable names possible, reuse variable names, use self-modifying code. Nor were these techniques theoretical, an abundance of program listings demonstrated them in real programs.

The ZX81 user manual [36] contains a section covering symptoms that a user might experience when memory is running out; but it does not give much in the way of concrete advice for reducing memory usage. A slew of third-party books quickly appeared for the ZX81 covering all aspects of the machine ranging from introductory BASIC programming through to electronic details of the hardware. Several contained sections on reducing memory usage [4, 14, 27, 22]. In addition many articles, letters, program listings, and other snippets dealing with memory appeared in a variety of magazines, some of which such as *Sinclair Programs*, *Sinclair User*, *Sync*, and *ZX Computing* were dedicated to the Sinclair computers. These early magazines catered better for the programmer than later magazines such as *Crash* and *Your Spectrum* (later *Your Sinclair*) which were more game focussed and dealt primarily with the ZX Spectrum. Many of these magazines can be found at the Internet Archive and the World of Spectrum websites. Articles dealing specifically with saving memory varied widely in accuracy and scope, but include [15, 13, 34, 7, 35, 3, 9, 38, 6, 8, 32].

Consider the BASIC statement `10 LET A=1`. On the ZX81 this line occupied 15 bytes: 2 bytes for the line number, 2 bytes for the line length, 1 byte for the token `LET`, 1 byte for the variable name `A`, 1 byte for the symbol `=`, 7 bytes for the number 1, and 1 byte for a trailing newline. It is immediately obvious that storing the number 1 is occupying a disproportionate amount of space. All the tokens of the language (`LET`, `IF`, `FOR`, `GOTO`, etc.) were stored efficiently as single byte elements of the ZX81 'character set'. Line numbers were encoded as a 2-byte big-endian form (allowable line numbers ranged from

1 to 9999, but other values, like 0, could be used if memory was directly modified). All other numbers appearing in numeric contexts in a BASIC program listing were stored as typed, followed by a hidden byte value (126) indicating the presence of a number, followed by a hidden 5-byte floating-point encoded value for the number. This allowed for faster execution, but means that writing a number n in a program actually uses $d(n) + 6$ bytes of memory, where $d(n)$ is the decimal string length of n. For example, 23 consumes 8 bytes while 3.14 consumes 10 bytes. Using this much memory to store a number was a significant burden on the 1 K ZX81.

Because both the visible and hidden representations of a number are set when a line is entered, it is possible to modify memory after the fact, to cause a different number to be evaluated than that visible in the program. For example, a program might contain the statement `GOTO 10`, but in fact jump to some other line number when actually executed. Another oddity is that an isolated period (`.`) is a valid representation for 0 on the ZX81. Therefore, it was possible to go through a program after it was written and compress out all but one of the visible digits in a number (and the last remaining digit could be replaced with `.` if desired).

The fact that each keyword in the BASIC language was a single byte in the ZX81 character set means that it is often possible to save space by using them in arbitrary strings. Further since using a keyword usually prints surrounding spaces, the saving can be even greater. McDaniel [28] gives some examples of this, including

```
100 PRINT " TO STOP PROGRAM, INPUT S"
```

where we have underlined the use of keyword tokens and free spaces provided by the keywords. By using the tokens `TO`, `STOP`, and `INPUT` a total of 12 bytes has been saved. A certain amount of creativity is needed to type some of these constructions because typing keywords typically requires the 'K' cursor. Other articles with this technique include [14, 35, 9].

Two general approaches to saving memory were based on the `CODE` and `VAL` functions.

The `CODE` function returns the numeric value of a character (or more precisely, of the first character in a string) [36, 14, 27, 13, 22]. For example, `CODE "*"` returns 23 on the ZX81. Because keywords like `CODE` occupy a single byte, this construction requires only 4 bytes (the space between `CODE` and `"` is implicit). This is a saving of 42.9% for one digit numbers, 50% for 2-digit numbers, and 55.6% for 3-digits numbers.

The `CODE` technique is limited to the character set so at best can produce integers in the range 0 to 255 inclusive. Some values of the character set cannot be typed on the keyboard but still can be used provided the relevant memory byte is directly modified (e.g. using `POKE`). Even so, there remain some problematic values: 11 (double quote), 118 (newline), and 127 (cursor) fail, and 126 (number) is marginal. Of these, 118 results in nonsense in BASIC, 11 and 127 evaluate to 0. While, 126 (number) can be `POKE`d in a running program and can work, it will typically result in an integer out of range error when the line is parsed.

A second mechanism applicable to all numbers including decimals uses the `VAL` function [36, 27, 22, 9]. The `VAL` function evaluates its string argument as a numeric expression. For example, `VAL "23"` (5 bytes) evaluates to 23. Writing `VAL "`n`"` therefore saves 3 bytes compared to the standard representation of n. However, as `VAL` can take any valid expression the savings can sometimes be larger: `VAL "1/8"` (6 bytes), `VAL ".125"` (7 bytes), `.125` (10 bytes), `1/8` (13 bytes). Similarly, `VAL "1E4"` (6 bytes) as a replacement for `10000 (11 bytes)`. The `VAL` function is capable of general expression evaluation and its argument may refer to variables. Suppose there is a need to generate random dice rolls in a program, then define the expression `LET A$="1+INT (RND*6)"` and thereafter generate random dice rolls with `VAL A$` [23].

Sinclair BASIC represents π with the `PI` keyword and thus a single byte. Actually, the ROM stores $\pi/2$, so using this constant does entail some additional computation [26]. The `PI` constant became the basis for an eclectic collection of representations for small integers. A selection of these constructions appears in Figure 1 along with timing. Values

for which the CODE technique is the best have been omitted. Most of the functions are self-explanatory, but Figure 2 is a brief summary. In Figure 1, PEEK is only use to look into the ROM. Parentheses around arguments to functions were not required in Sinclair BASIC. Negative values can be obtained by adding one additional byte (-) at the front of the corresponding positive value. Where there are better ways of getting negative values these are indicated in the table. Sometimes a floating-point value can be used a proxy for an integer; GOTO PI will jump to line 3, thus in this case a single byte successfully represents 3.

There was no hidden memory cost in referring to existing variables, so if two or more variables need to be initialized to the same value, it makes sense to initialize subsequent occurrences from the first one: 10 LET A=1, 20 LET B=A, etc., and similiarly you could get a 2, using A+A (3 bytes, 157 FRAMES).

n	b	time	expression		n	b	time	expression	
–	–	100	REM		9	6	226	INT (PI*PI)	[37]
-5	8	132	-5		11	8	124	11	
-5	4	4136	INT TAN SQR PI		11	5	169	PEEK CODE "X"	
-5	5	154	-CODE "▌"		11	5	308	VAL "11"	
-5	5	238	-VAL "5"		15	4	2380	INT SQR PEEK PI	
-5	5	242	VAL "-5"		20	4	1021	INT EXP INT PI	
-4	3	227	INT -PI		22	4	2397	CODE STR$ -PI	
-1	2	1013	COS PI	[35, 9, 8]	22	4	2405	PEEK LEN STR$ PI	
0	7	123	0		23	3	975	INT EXP PI	[31, 10]
0	2	150	NOT PI	[15, 24, 27, 25]	27	5	2388	INT PI**INT PI	
0	2	385	INT RND		28	4	226	CODE STR$ NOT PI	
0	2	952	SIN PI	[16, 31, 1, 39]	29	4	2389	CODE STR$ SGN PI	
0	2	1831	TAN PI		30	4	2898	CODE STR$ PEEK PI	
0	3	143	CODE ""	[11]	31	3	2379	CODE STR$ PI	[9]
0	3	143	LEN ""		54	4	213	PEEK PEEK PEEK PI	
0	3	199	PI-PI	[4, 12, 33, 21]	118	9	125	118	
1	2	151	SGN PI	[15, 24, 25]	118	5	168	PEEK CODE "."	
1	3	171	PI OR PI		118	6	387	VAL "118"	
1	3	203	PI=PI		122	5	4019	INT EXP SQR EXP PI	
1	3	215	PI/PI	[4, 27, 12, 33]	126	5	170	PEEK CODE "."	
2	4	146	CODE "▗"		127	5	3239	PEEK INT SQR EXP PI	
2	4	1008	INT EXP SGN PI	[5]	135	3	189	PEEK PEEK PI	[31, 9]
3	2	163	INT PI	[31, 10, 5]	195	4	3216	PEEK SQR EXP PI	
4	4	236	-INT -PI		203	4	3179	PEEK EXP SQR PI	
4	4	237	ABS INT -PI		209	4	197	PEEK PEEK NOT PI	
4	4	3214	INT SQR EXP PI	[31, 5]	211	3	173	PEEK NOT PI	[9]
5	4	1516	INT LN PEEK PI		211	3	975	PEEK SIN PI	[31]
5	4	3177	INT EXP SQR PI		253	3	176	PEEK SGN PI	[9]
9	3	2379	LEN STR$ PI	[9]	255	2	165	PEEK PI	[31, 9]
9	5	243	INT PI*INT PI						

Figure 1: Memory saving representations, n is the number, b is the number of bytes needed for the expression.

The timing numbers in the table were computed with the sz81 emulator [2] using the BASIC program in Figure 3. In each case line 40 was replaced with a statement of the form 40 LET A=*expression*. The actual timing is measured via the FRAMES system variable [36]. These timings prove remarkably consistent with multiple runs of each expression invariably producing exactly the same timing. The design of sz81 ensures the timings are an accurate reflection of the timing on a real ZX81.

It is hard to pinpoint when any of these constructions were first noticed, but up to four citations of early occurrences have been given. Likely many of the tricks were discovered

Keyword	Meaning
`CODE`	ZX81 character code of first byte of string (or 0 for empty string).
`INT`	Floor function.
`LEN`	Length of a string.
`NOT`	Logical not, any nonzero argument gives 0.
`PEEK`	Content of memory at the given address.
`PI`	$\pi = 3.14159\ldots$.
`RND`	Pseudorandom number r in the range $0 \leq r < 1$.
`SQR`	Square root.
`STR$`	String representation of its numeric argument.
`VAL`	Evaluate string argument as an expression.
`**`	Powers, `2**3` $= 2^3 = 8$.

Figure 2: Brief guide to some ZX BASIC functions.

```
5 CLEAR
10 POKE 16437,255
20 POKE 16436,255
30 FOR N=0 TO 99
40 REM
50 NEXT N
60 PRINT 65536-PEEK 16436-256*PEEK 16437
```

Figure 3: BASIC program to time number constructions.

multiple times and passed by word of mouth at various user groups. Indeed, we determined many of the entries in Figure 1 before finding references to them in the literature. Even after better or faster representations appeared other forms continued to be used. For instance, the forms `PI-PI` and `PI/PI` remained popular long after the `NOT PI` and `SGN PI` forms first appeared.

References

[1] ? Math maze. *Sinclair User*, page 30, October 1982.

[2] sz81. `http://sz81.sourceforge.net/`, 2016.

[3] David Anderson and Ian Morrison. Easy ways of getting quart into a pint pot. *Sinclair User*, pages 50–51, February 1983.

[4] Beam Software. *The Complete Sinclair ZX81 Basic Course*. Melbourne House, Leighton Buzzard, Bedfordshire, 1981.

[5] Olly Betts. The art of writing small programs. *Open Source Developers' Conference*, November 2011. `https://survex.com/~olly/talks/small-programs/small-programs/`.

[6] Dave Cartwright. Save memory. *Sinclair Programs*, page 7, December 1983.

[7] John Coffey. Memory saving tips. *Sync*, 2(5):6, September/October 1982.

[8] M. J. Davies. 3 into 1K goes. *Sinclair Programs*, pages 19–20, October 1984.

[9] James Grosjean. Memory scrunching on the TS1000 and ZX81. *Sync*, 3(5):80–84, September/October 1983.

[10] Tim Grubb. Escape. *Sinclair User*, page 67, March 1983.

[11] Said Hasson. Alley driver. *ZX Computing*, page 72, August/September 1982.

[12] D. E. Healey. Enterprise rescue. *Your Computer*, page 71, March 1982.

[13] Andrew Hewson. Making the best use of memory. *Sinclair User*, pages 57–58, June 1982.

[14] Andrew D. Hewson. *Hints & Tips for the ZX81*. Hewson Consultants, June 1981.

[15] Kevin Hill. Memory thrift. *Your Computer*, page 11, November 1981.

[16] D. G. Hockey. Dragon crunch. *Sinclair Programs*, page 9, July/August 1982.

[17] David Horne. 1K ZX chess. Sinclair, 1982.

[18] David Horne. Chess 1K. *Your Computer*, pages 68–69, December 1982.

[19] David Horne. Chess 1K. *Your Computer*, page 81 and 83, January 1983.

[20] David Horne. Chess in 1K. *Your Computer*, pages 100–102, February 1983.

[21] I. S. Howson. Minefield. *Sinclair User*, page 31, May 1982.

[22] M. James and M. Gee S. *The Art of Programming the 1K ZX81*. Bernard Babani, London, July 1982.

[23] Dilwyn Jones. Hints 'n' tips to improve your programs. *ZX Computing*, pages 100–102, August/September 1982.

[24] Ian Logan. Understanding floating-point arithmetic: Part 1. *Sync*, 2(1):30–32, January/February 1982.

[25] Ian Logan. Understanding floating-point arithmetic: Part 2. *Sync*, 2(2):18–22, March/April 1982.

[26] Ian Logan and Frank O'Hara. *The Complete Timex TS1000 & Sinclair ZX81 ROM Disassembly*. Melbourne House, 1982.

[27] Mike Lord. *The Explorers Guide to the ZX81*. Timedata Ltd., Basildon, Essex, February 1982.

[28] Richard W. McDaniel. Using key and token expressions. *Sync*, 1(5):29, September/October 1981.

[29] M. Douglas McIlroy. Development of a spelling list. *IEEE Trans. on Communications*, 1(1):91–99, 1982.

[30] Psion Software. Computer scrabble. Sinclair Research, 1983.

[31] Philip Pulsford. Small enterprise. *Your Computer*, page 83, October 1982.

[32] Tony Rickwood. Program to achieve speed and efficiency. *Sinclair Programs*, pages 34–35, November 1984.

[33] Tim Rogers. Star swerver. *Sinclair User*, page 32, April 1982.

[34] D. J. Todorovic. Getting a quart into a pint pot. *Sinclair User*, pages 24–25, August 1982.

[35] Roger Valentine. What can i do with 1K. *ZX Computing*, pages 91–92, February/March 1983.

[36] Steven Vickers. *Sinclair ZX81 BASIC Programming*. Sinclair Research Limited, 2nd edition, 1981.

[37] D. Watts. Tank attack. *Sinclair Programs*, page 33, March/April 1983.

[38] P. Williamson. Save memory. *Sinclair Programs*, page 7, November 1983.

[39] R. J. Zealley. Towers of Hanoi. *Sinclair Programs*, page 24, March/April 1983.