Atomic Basic Specification

Written by Paul Robson December 2020

Introduction

This is not the manual.

Terms

These terms are supported

Example	Function	Description
42	No	32 bit decimal
&422a	No	32 bit hexadecimal
a.name	No	Variable, names which consist of alphanumeric characters and a period, the first character must be non numeric.
array[index]	No	Array access. Arrays do not exist as such – a variable points to a block of memory.
@<reference>	Yes	Converts reference term to address, used to get the address of variables, which can be used to pass by reference.
“hello, world !”	No	Evaluates to an ASCIIZ string (e.g. ending in zero)
-<term>	No	2’s Complement Negation
~<term>	No	1’s Complement
!<term>	No	Word indirection (a 4 byte PEEK)
?<term>	No	Byte indirection (like PEEK)
$<term>	No	Converts constant to string, value unchanged, this types it as a string.
(<expr>)	Yes	Parenthesised Expression
function(param)	Yes	Functions (various)
True	Yes	Constant
False	Yes	Constant

Binary Operators

The following operators are supported

Precedence	Operators	Descriptors
1	and or xor	Binary operators
2	>= <= = <> > <	Comparison, on either string/number
3	+ -	Additive
4	* / >> << %	Multiplicative (shifts are unsigned)
5	! ? $	References. These operate like the unary operators on the sum of the left and right hand side, so A?4 is the same as PEEK(A+4)

L-Expressions

The following references are used, on the left hand side of an assignment statement.

Type	Description
count	A variable
count!4 count?4	Byte/Word indirection. So Count?4 = 2 is the same as POKE Count+4,2
!count ?count	Byte/Word indirection without the addition (e.g. ?Count=1 is the same as Poke Count,1)
test(4)	Array element. Arrays don't exist as objects in their own right ; an array is a variable pointing to an area of memory, which is allocated at 4 bytes per element, e.g. this will be offset 16..19 from the address in test.
$str str$5	These can be used on the left hand side but they operate like StrCpy in C. So if you write (say) $Str = "Hello world" and Str has the value 5000 (everything being 32 bit integers), the value of the right side (the address of the 'H' character in the string) is a source for a string to be copied to the value of the left (5000) – so the code for 'H' is copied to 5000, 'e' to 5001 and so on, through to 5011 having a zero stored in it. If you write str = "Hello World" the address of the H is stored in the variable str.

Tokens

Token	Value
00-2F	Identifier end markers. (00-19 = A-Z, 1A-29 = 0-9 2A = .
30-5F	Identifier ‘in line’ markers
60	ASCIIZ string follows. The low byte is the offset from the start of the token (60) to the byte after the trailing zero, limiting strings to 252 characters
61-6F	Reserved
70-7F	Integer, representing 4 bits. These can be followed by other integers which occupy successively higher nibbles in the integer, so the sequence 74 72 7C represents the hex value $C24
80	End of line token
81-FF	Other tokens. Each token has a type table entry. Commands are 80-82. Binary operators are 00-0F. Unary functions are 40. Multiple use (e.g. -, $) are defined as binary and handled seperately.

Storage

Programs are stored in consecutive memory. Above the last marker (offset $00) are stored variables and allocatable memory.

Offset	Data
0	Offset to the next line in bytes, or $00 if end of program.
1	Line number low
2	Line number high
3	First token
4 ….	More tokens
….	$80 end of line marker

Variables

Variables are kept in an array of linked lists. The hash for the list is the xor of the second character and the first character arithmetically shifted left. (as a-z are stored seperately). The number of linked list is currently 16 but can be varied and be any power of 2 from 1 to 128, save for both the A-Z variables and the hash table must be on the same page.

Offset	Contents
+0,+1	Address of next variable name or $0000 if end of list
+2	Full 8 bit hash of variable.
+3,+4	Address of variable name in tokenised form (comes from creation)
+5..+8	Variable value

Procedures

When the program is first run it builds a linked list of procedures, which it can check when invoking. The hash for this is the total of *all* the character codes in the name, modulus 256.

Offset	Contents
+0	Offset to next procedure, or zero if the end of the list.
+1	Full 8 bit hash of the procedure
+2,+3	Address of the procedure name start
+4,+5	Address of the procedure code (e.g. after the closing parenthesis)
+6	No of parameters
+7,+8	Address of parameter #1
+9,+10	Address of parameter #2

Built in "AMORAL Junior".

The language built in is a loose version of AMORAL. The current register is now YX not XA. It is 16 bit unlike the 32 bit BASIC.

Command	*Purpose*	*Code*
(cv)	*Load Register*	*Ldx / Ldy address*
+ - and or xor (cv)	*Binary operation.*	*Txa / Op / Tax* *Tya / Op / Tay*

		Hard coded SEC/CLC *Hard coded 0 optimisations.*
** / % (cv)*	*Binary operations*	*Subroutines with the address or constant coded in the bytes following.*
>= < = <> (cv)	*Compare*	*Does cmp/subtract or xor and sets the compiler branch flag to BEQ/BNE/BCC/BCS “true” opcodes.*
->(var)	*Save Register*	*Stx / Sty address*
#(var)	*Set current array*	*Load address const to zp*
[(const)]	*Load array element*	*Use Y to load on zp*
->[(const)]	*Save array element*	*Use Y to save on zp (ideally non destructive, think about this one)*
While (<code>) …. Wend	*While Loop*	*Compile code, branch out on opcodes*
If (<code>) …. Else …. Endif	*If decision*	*Compile code and skip over, else is optional*
(var)()	*Call*	*JSR variable address.*
++ – >> <<	*Unary operations*	*Appropriate code.*

Dictionary Format

Offset	*Contents*
+0	*Offset to Next entry to be matched against, or zero.*
+1	*Keyword token to match against ($FE variable $FF constant)*
+2	*No operand (0) Constant operand (1) Variable operand (2) Either (3)* *Bit 7 specifies 'hard coded'*
+3	*Byte count following (output code may be larger or smaller)*
+4	*Byte data #1*
+5	*Byte data #2*

Where "hard coded" is specified, locations 3 and 4 are the address of the code to execute.
Fill in values

These substitution values are used when outputting code

Hex	*Element*	*Notes*
F3	*Low value const*	
E3	*High value const*	
D3	*Low Value * 4*	*Used for arrays*
C3	*Address.W*	*Outputs 2 bytes*
B3	*Address.W+1*	*Outputs 2 bytes*
93	*Output Opcode ...*	*Outputs 1 byte, previously setup.*
83	*Set Opcode to ...*	*Following byte sets the 'working byte'.*