# RPL/C

*Written by Paul Robson January 2020*

RPL/C is an interpreted FORTH/BASIC cross, which borrows implementation ideas from FORTH and BASIC. It is an attempt to import the speed of FORTH into an Interpreted environment.

## Line format

The line format is the classic one.

1. An offset to the next line, which can be zero indicating the program end
2. The line number in low/high order
3. The encoded line data

## Encoded Line Data

The encoded line data borrows the implementation idea from FORTH. It is a list of 2-byte addresses and data that is executed using the INX INX JMP ($nnnn,X) sequence borrowed from FORTH. The implementations of this design (that X is < 256) doesn’t matter because of the limits on the length of the line.

## Types of Code elements

|  |  |  |
| --- | --- | --- |
| Type | Param | Notes |
| **standard** | None | Ordinary 65C02 words that have a specific function – dup list new ; @ c@ etc that exit with JMP NEXT |
| **constants** | None | Words that are common constants – 0 1 2 etc. so that they do not have to be done using the “Literal” option. |
| **$$literal** | 2-byte integer | Used for uncommon constants, that push the two-byte integer parameter on the stack. |
| **$$call** | 2-byte line number | Call the given line number, which should be a definition. If it isn’t, the syntax <number> is used instead, encode should cope with either syntax. |
| **$$define** | Name data | Define a word (used for :name). The name should contain the overall length in the first byte of the name, be in upper case, have the last character with bit 7 set so ABCD would be $05 $41 $42 $43 $C4. For :definition |
| **$$comment** | String data | Comment which has the same format as $$define but uses up the rest of the line including spaces. |
| **$$string** | String data | Push a string address onto the stack, encoded as for $$comment, except that it should end with a $00, making the string ASCIIZ. |
| **$$nextline** | None | This is the last token on each line, and it transfers execution to the *next* line. |
| **if else until for next** | Branch position (1 byte) | These are used for various loops and tests. Loops and tests must be completed on the same line. This is because lines are stand-alone entities. This can be a good thing, no *big* loops or conditionals. |
| **repeat endif** | None | These are dummies, that do nothing but are required for decompilation. |
| **@variable**  **!variable**  **&variable** | Name | This compiles a standard $$@handler $$!handler $$&handler worker function which reads, writes or returns the address of the variable. The name is composed by first + second x 32 + third x 32 x 40. The first character must be A-Z, the second can be A-Z or 0-9. |
| **[n]** | Index  (1 byte) | This compiles $$index followed by a byte. This is purely a marker and can only follow a variable definition. n can be 0-127 only and must be a constant. It converts the variable into an array, where the base address of the array is the variable value. |

## Dictionary Element

The dictionary does *not* expand, though it may well be possible to add libraries later.

|  |  |
| --- | --- |
| **Offset** | **Contents** |
| +0 | Offset to next entry, 0 indicates the end of the list. |
| +1 | Type byte  Bit 7 : Set if this word is not displayed  Bit 6 : This word cannot be run from the command line  Bit 5 : Set if there is a special encoder for this word  Bit 4 : Set if there is a special decoder for this word  Bit 3,2: Set to zero  Bit 0,1: Parameter count (0,1,2,n) |
| +2,+3 | Code address to execute this word. |
| (Optional) | Decode address if decode-special bit is set |
| (Optional) | Encode address if encode-special bit is set. |
| Name | The name in upper case with bit 7 of the last character set. Note that this position will vary depending on the bits in the type byte. |

The optional addresses depend on bits 4 and 5 ; so, the name can be at offset 4,6 or 8 depending on those bits.

It is a convention that words beginning with $$ are special and not decoded normally, but this is not enforced. The Vlist command does not show them.

If encoders exist, they are called when the word is tokenised – normally a word is simply tokenised by having its call address compiled. These are equivalent to immediate words in FORTH. So, if, for example, might stack the address or offset of the branch.

If decoders exist, they are called when the word is listed – so for example the 2-byte literal word will display the literal as an integer, reading it from the next 2 bytes. Some (like IF) will skip data while still displaying the token word ‘if’.

## Dictionary Markers:

Dictionary markers use the *<label> : ;; <contents>* format but extended to identify words and handlers. The contents are the word in question , followed by a series of options which are *encode decode* (sets encode and decode for this word, defaults to none/standard) *hide noexec* which set the type byte bits. If no option is provided it sets the code execute address as normal. Note that encoders and decoders should be in the same file *after* the words code definition.

## Execution

The execution design is the inx jmp ($nnnn,x) as described earlier. TOS is kept in a separate memory location. The 6502 return stack is used for the data stack, and an indexed stack used for return addresses.

## Enhancements

[x] [n] array options

Fast look up of line numbers