

UPL

Utility Programming Language for 6502 Processors.

Be afraid. Be very afraid.

VIC-20 Implementation.

Document: KD-UPL-UM-01.

This Document © Brendan Jones, 1994, 1998. All Rights Reserved.

As of April 12, 1998 the UPL Software is in the public domain.
Refer to the file LEGAL.TXT distributed with the software.

Legal Notice.

This legal notice covers the terms of use of this document. As of April 12, 1998 The Commodore VIC-20 and Apple II implementations of the UPL Software have been placed in the public domain. Refer to the file LEGAL.TXT distributed with the Software. The software should not be distributed without that file.

Before you use this document for the first time, please read this Disclaimer and License Agreement. By your action of using this document, you are agreeing to comply with all terms contained in this License.

LIMITATION OF LIABILITY: WE (BRENDAN JONES AND OUR AUTHORISED SUPPLIERS) OFFER NO WARRANTY OF ANY KIND EITHER EXPRESSED OR IMPLIED INCLUDING THOSE OF MERCHANTABILITY, NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY, OR FITNESS FOR A PARTICULAR PURPOSE. NEITHER WE NOR OUR AUTHORISED SUPPLIERS AND DISTRIBUTORS SHALL BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE OUR SOFTWARE, PRODUCTS OR SOFTWARE, EVEN IF WE OR OUR AUTHORISED SUPPLIERS AND DISTRIBUTORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THIS PRODUCT OR SERVICE MAY NOT BE USED IN JURISDICTIONS THAT PROHIBIT THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES.

You may redistribute this documentation, providing you meet the following conditions: a) The document is distributed as a whole, including without modification this legal notice. b) You do not receive direct or indirect financial gain from the act of distributing or copying this document.

This document contains other trademarks and servicemark names. These are the property of their respective owners. Brendan Jones has placed the name "UPL" in the public domain.

References in this publication to products, programs, or services do not imply that we intend to make these available in all countries in which we operate.

This publication may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

We welcome comments on products and services including this software and documentation. We may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Table of Contents.

1. INTRODUCTION.....	7
1.1 ACKNOWLEDGMENTS.	8
2. THE UPL PACKAGE.....	9
2.1 RUNNING UPL ON THE VIC-20.	9
2.1.1 Loading from Tape.	9
2.1.2 Using UPL with a VIC-20 Emulator.	9
3. THE UPL LANGUAGE.....	11
3.1 IDENTIFIERS.	11
3.2 DATA TYPES.	11
3.3 PROGRAM.	11
3.3.1 Constants.	12
3.3.2 Variables.	12
3.4 STATEMENTS.	13
3.4.1 Assignment Statement.	13
3.4.2 inc Statement.	13
3.4.3 dec Statement.	13
3.4.4 put, putln Statements.	14
3.4.5 field Statement.	14
3.4.6 get Statement.	14
3.5 STRUCTURED STATEMENTS.	15
3.5.1 Compound Statement.	15
3.5.2 if Statement.	15
3.5.3 repeat Statement.	15
3.5.4 while Statement.	16
3.6 EXPRESSIONS.	16
3.7 SUBROUTINES.	17
3.7.1 Procedure.	17
3.7.2 Function.	17
3.8 ADVANCED STATEMENTS.	18
3.8.1 goto Statement.	18
3.8.2 err Statement.	18
3.8.3 push and pop Statements.	19
3.8.4 clear Statement.	19
3.8.5 mem Array.	19
3.8.6 call Statement.	20
3.8.7 mach Statement.	20
3.9 COMMENTS.	21
3.10 EXAMPLE.	21
3.10.1 Multiplication Table.	21
4. IMPLEMENTATION: COMMODORE VIC-20.....	22
4.1 PACKAGE CONTENTS.	22
4.2 THE MENU.	22
4.2.1 New.	22
4.2.2 Read text.	22
4.2.3 Save text.	23
4.2.4 List.	23
4.2.5 Modify.	23
4.2.6 Delete.	23
4.2.7 Insert.	24
4.2.8 Quit.	24
4.2.9 Object save.	24
4.2.10 eXecute.	25
4.2.11 Compile.	25
4.3 COMPILING.	27

4.4 ERRORS.....	27
4.4.1 ?Out of Memory error.....	28
4.5 ERRORS.....	28
4.5.1 System Errors.....	28
4.5.2 Runtime Errors.....	29
4.5.3 Compile Errors Codes.....	29
4.6 MEMORY CONFIGURATION.....	30
4.7 LOOK MA! NO COMPILER!.....	31
4.8 ADVANCED FEATURES.....	32
4.8.1 Suppressing Overflow.....	32
4.9 BUGS.....	32
4.10 EXAMPLE.....	33
4.10.1 VIC-20 Meteor Shield.....	33
5. IMPROVEMENTS.....	36
6. EMULATORS.....	38

1. Introduction.

Be afraid. Be very afraid. Did you know that in 1983 the Commodore VIC-20 had a full-blown Pseudo-Pascal compiler? It generated native 6502 machine code. Although the compiler was slow and had the unfortunate limitation of byte variables, the unoptimised machine code it produced was very fast. The language was dubbed UPL; “Utility Programming Language.” You could do anything in UPL you could do in assembler. UPL was never released commercially. By 1984 the VIC was well on its way to extinction, being overtaken by the vastly superior (and incompatible) Commodore 64.

Many years later, nostalgic for the arcade games of the good ‘ole days some talented programmers have written VIC-20 emulators. These emulators allow old software written for the VIC to be run on nearly any modern PC. (See the appendix for a list of emulators and where you can download them.)

So now nearly fifteen years after I wrote UPL I’m able to rerelease purely as a historical curiosity; An example of what you can do on a limited machine with limited resources if that’s all you’ve got. It’s also a rebuttal to the revisionists who view think the VIC was a practical joke by Commodore. The UPL Compiler itself ran entirely within the 16Kb expansion cartridge. The VIC’s unexpanded 3.5Kb was left alone for storage of the compiled UPL programs and the runtime library. The expanded VIC was considered the development system, and the unexpanded VIC the target system for UPL programs. This document is largely taken from the original 1984 manual, typed up on a DEC KL-10 mainframe! Emulator hints are printed in blue.

I wouldn’t recommend developing anything new with UPL. You could do much better with a 6502 C cross-compiler. Then again you could do better still with a Pentium native compiler. ☺ However if you want a laugh you can get it to say “Hello World” and look at the arcade game listed beginning on Page 33. (Yes, It really does work.)

If you get a kick out of UPL then visit www.kdef.com/geek/vic. I’ve loaded up the best of my old programs, including some still-playable arcade games and adventures. Enjoy!

Brendan Jones.

April 12, 1998.

E-mail: bj@kdef.com

Web: www.kdef.com/geek/vic

1.1 Acknowledgments.

- The authors of the many VIC20 emulators for saving the VIC20 and its software from oblivion. (They're listed on Page 33.)
- Nikolaus Strater (nstrater@mcmail.com) for the VTR VIC-20 to PC tape loader. Nikolaus says the number of people wanting to use this program worldwide must be frighteningly small; perhaps 2-3. Make that 4!
- Jeff Minter (Author of Gridrunner) for starting the ball rolling by freeing up his own commercial VIC-20 software.

2. The UPL Package.

The UPL package consists of three programs. These programs have different names on the 1984 tape and 1998 emulator release of UPL.

File.	Description.
Bootstrap.	The UPL Boot program starts the loading process. It configures your VIC moving the video memory to the same location it appears on an unexpanded VIC. You'll thus be able to develop UPL applications using the same memory configuration as your target system. The boot program then loads the UPL Runtime library and then loads and runs the UPL Compiler. (On the original 1984 tape version of UPL this program was called "UPL BOOT.B16". On the 1998 rerelease for emulators it is now called "UPL-BOOT.BAS.")
Runtime Library.	This is the UPL Runtime Library. It is a binary file loaded between memory locations 4105 and 5008, inclusive (\$1009 and \$1390 hexadecimal). This file must be present for any UPL programs to run. You can load the runtime library without the boot program simply by typing the BASIC command LOAD "UPLRTIME.BIN" or LOAD "RUNTIME.OBJ". (On the original 1984 tape version of UPL this program was called "RUNTIME.OBJ". On the 1998 rerelease for emulators it is now called "UPLRTIME.BIN.")
Compiler/Editor.	This is the UPL Compiler and Editor. Use this program to create, edit and compile UPL programs. (On the original 1984 tape version of UPL this program was called "COMP/EDIT.B16". On the 1998 rerelease for emulators it is now called "UPL-TAPE.BAS.")

2.1 Running UPL on the VIC-20.

2.1.1 Loading from Tape.

The UPL Package requires a VIC-20 with at least 16Kb of expansion memory.

Turn your VIC-20 on. Insert the UPL Package cassette into your VIC-20's tape drive. Holding down the SHIFT key tap the RUNSTOP key. You will be asked to press PLAY on your tape drive. The full UPL package will be loaded automatically. When loading is completed you will be presented with the UPL development menu.

2.1.2 Using UPL with a VIC-20 Emulator.

You may run the UPL Package on any modern microcomputer with a VIC-20 emulators. Emulators are available for DOS, Windows, Unix and the Amiga. Refer to the Appendix for a list of available emulators.

With the PCVIC emulator you may load UPL with a single command in DOS:

```
PCVIC UPL-TAPE.PCV
```

With the V20 emulator you may load UPL by first typing in DOS:

```
V20
```

Then select the “load state file” option from the “machine” pulldown. Enter the name “UPL-TAPE.V20”. UPL will then be loaded.

To load UPL on any other emulator follow these instructions:

1. Load and run the program “UPL-BOOT.BAS”.
2. It will reconfigure your VIC in preparation for loading the rest of UPL. It’ll ask you to press the PLAY button on your tape drive. Of course the vast majority of emulators don’t have a tape drive. Press the RUNSTOP key on the emulated keyboard. (You’ll have to refer to the emulator documentation to find which key is acting as RUNSTOP. On PCVIC it is NUMLOCK. On V20 it is TAB.)
3. Load the binary file “UPLRTIME.BIN”.
4. Load the BASIC program “UPL-TAPE.BAS” (If you’re using PCVIC you’ll have to tell the emulator to “undelete the BASIC program” after doing this).
5. Type the BASIC command “RUN” and press Return (Enter).
6. You should now be at the UPL menu.
7. At this point we recommend taking a “system snapshot” with your emulator. You can use the snapshot to start UPL quickly without having to go through the above steps.

Most emulators lack emulated tape and disk drive support. This means you must type in your UPL programs from scratch. However it will be possible to save and run your compiled programs separately. See Section 4.2.9 on Page 24. You can also save your UPL source code by telling the emulator to save the entire VIC-20 in a system “snapshot” file. Refer to your emulator documentation for more detail.

3. The UPL Language.

3.1 Identifiers.

Constants, Variables, Procedures and Functions are all identified by a name. This name is known as an “identifier.” An identifier consists of a single lower case letter followed by zero or more lower case letters or digits. No two objects may have the same identifier.

For example:

```
people
counter
plan9
californiastatesalestax
```

3.2 Data Types.

UPL recognises only one data type; the byte. A byte may hold an integer between the values -128 and 127, inclusive.

With a compiler option you may specify that integers the range of values between 0 and 255 inclusive is also accepted. When this happens the values between -128 and -1 are mapped onto 128 and 255 using the principle of “two’s compliment.” (Refer to a computing text on binary arithmetic for further information.) Even if you enable this compiler option you’ll still need to patch the runtime library to suppress overflow errors when calculating with these larger numbers.

3.3 Program.

A UPL Program has the following form:

```
[ constant-declarations ]
[ variable-declarations ]
[ subroutine-declarations ]
compound-statement
.
```

The compound-statement is the main body of the program. It is what is executed when the program starts running. This compound-statement must be followed by a single full-stop (also known as a period) “.”. Those clauses shown in square brackets are optional. They may be omitted if there are none to declare. Do not confuse these square brackets with the bold square brackets of a compound statement; eg. “[]”. The bold square brackets indicate that the square brackets appear in the UPL source code.

Here is a sample UPL program:

```
[putln("Hello World");].
```

3.3.1 Constants.

A constant may have a value of any integer ranging between -999,999 and 999,999, inclusive. Unlike a variable a constant's value may not be modified outside of its declaration. If you assign or otherwise pass a constant to a variable or an expression only the lowest 8 bits are transferred. For example, if the a constant having a value of 259 is assigned to a variable then the variable will hold a value of 3.

The constants declaration has the following form:

```
cons  
  { identifier = integer-value # , } ;
```

The bold word **cons** and characters "=", ",", and ";" indicate they appear literally within the source code. The curled braces indicate that the enclosed clause may be repeated zero or more times, so long as a comma "," is used to separate each instance.

For example:

```
cons  
  numpeople=9,  
  carriagereturn=13,  
  limit=16,  
  clearscreenroutine=58901,  
  amountowed = 64;
```

3.3.2 Variables.

The variables declaration has the following form:

```
var  
  { identifier # , } ;
```

For example:

```
var  
  count, sum,  
  person6;
```

The initial value of a variable is arbitrary; It should not be assumed to be zero.

The variables **x**, **y**, **a** and **p** are automatically declared. These correspond (although they are not actually) the 6502 processors registers. **c** is a pseudo-variable that is true (non-zero) if

the processor's carry flag is set. The variable **return** is automatically declared in functions to receive the value return as the function's result.

3.4 Statements.

3.4.1 Assignment Statement.

An assignment statement assigns an expression to a variable. Assignment statements have the following form.

identifier = integer-expression

Here are some examples:

```
numpeople=35
count=count + 1
x = 2 *(numberin - y) + k / 6
```

Spacing between the elements of an expression are ignored.

3.4.2 inc Statement.

The **inc** statement increments the variable of a variable. That is, it adds one to it. If the variable is 127 then the value will wrap around to -128. Here is the form of the **inc** statement:

inc identifier

For example:

```
inc count
```

3.4.3 dec Statement.

The **dec** statement decrements the variable of a variable. That is, it subtracts one from it. If the variable is -128 then the value will wrap around to 127. Here is the form of the **dec** statement:

dec identifier

For example:

```
dec count
```

3.4.4 put, println Statements.

The **put** and **println** statements both print the value of an expression or a string of characters. To print an expression as an integer follow it by the hash character “#”. To print it as an ASCII character omit the hash. Here is the form of the **put** statement. The only difference the **put** and **println** statements is that **println** finishes by printing a newline character.

```
put    ( { expression [ # ] | character-string # , } )  
println [ ( { expression [ # ] | character-string # , } ) ]
```

The vertical bar indicates that each clause may be either an **expression** or a **character-string**.

For example:

```
put(33)                <prints ASCII character 33; "!">  
put(33#)               <prints the number 33>  
put("hello there")    <prints "hello there">  
println("hello there") <prints "hello there" and goes to a newline >  
put("The answer is ", 10+15#) <prints "The answer is 25" and newline>
```

3.4.5 field Statement.

The **field** statement specifies the number of characters that shall be used to print a number. If this the field is set to zero, then only the minimum number of characters needed to print the number will be used.

```
field ( expression )
```

For example:

```
field(0); put(5#, 12#); field(10); println(44#, -127#);
```

For the purpose of illustration we'll use a dot to represent each space in the output:

```
5-12.....44.....-127
```

Note in the above example we use a semicolon “;” to separate the different statements. Statements appearing in a list (except for structured statements, discussed later) must be separated by semicolons.

3.4.6 get Statement.

The **get** statement gets values from variables from the input device (usually the keyboard). If the variable name is followed by a “#” it will be retrieved as an integer, specified one per line. If the variable name is not followed by a “#” then a single ASCII character will be fetched from the input device and stored in the variable.

get ({ variable-name [#] # , })

For example:

<code>get(key)</code>	<gets a single character and stores it in variable key>
<code>get(count#)</code>	<gets a single number and stores it in variable count>
<code>get(x#, y#)</code>	<gets two numbers on two lines; one for x, one for y>

3.5 Structured Statements.

3.5.1 Compound Statement.

A compound statement groups zero or more statements together as a single statement. Each statement is separated by a semicolon. The last statement should not be followed by a semicolon.

[{ statement # ; }]

For example:

```
[ putln("Another one bites the dust"); inc count ]
```

3.5.2 if Statement.

The **if** statement lets you conditionally execute one or two statements. If the expression is true (non-zero) then the **then** statement is executed. If it is false then the **else** statement is executed instead.

if expression **then** statement [**else** statement]

For example:

```
if count = 2
then putln("two")
else putln("it is ", count)
```

3.5.3 repeat Statement.

The **repeat** statement repeatedly executes a list of statements until an expression becomes true (non-zero). The **repeat** statement is post-tested, which means the expression is evaluated after the statements have been executed. This means the statements are executed at least once.

repeat { statement # , } **until** expression

For example:

```
count = 1;
repeat
    println(count);
    inc count
until count = 11;
```

3.5.4 while Statement.

The **while** statement repeatedly executes a statement until an **expression** becomes false (zero). The **while** statement is pre-tested, which means the **expression** is evaluated before the statement is executed. If the expression begins false then the statement will not be executed even once.

while expression **do** statement

For example:

```
count = 1;
while not(count = 11) do
[
    println(count);
    inc count
]
```

3.6 Expressions.

Expressions may be made of the following operators.

Operator.	Description.
not expression	Returns the logical negation of the expression. eg. not 0 returns -1, and not -1 returns 0.
expression and expression	Returns true if both expressions are true.
expression or expression	Returns true if either expression is true.
expression = expression	Returns true if the two expressions are identical.
expression * expression	Multiplies two integers together.
expression + expression	Adds two integers together.
expression - expression	Subtracts the second integer from the first.
expression / expression	Divides the first integer by the second, returning a whole number.
expression ! expression	Divides the first integer by the second, returning the remainder.
- expression	Returns the arithmetic negation of an integer expression. eg. -2
pos factor	Returns true if a factor ≥ 0 .

neg factor	Returns true if a factor < 0.
zero factor	Returns true if a factor = 0.

UPL represents true by the value -1 and false by the value 0.

The usual operator precedence is applied. From strongest to weakest precedence these are:

- **and, or**
- ***, /, !**
- **+, -**
- **not, neg, pos, zero**

Brackets may be used to override operator precedence.

3.7 Subroutines.

A UPL program may have any number of functions and procedures.

3.7.1 Procedure.

A procedure is a subroutine that does not return a value. It may be called from any place a statement may be called. Procedures do not have any parameters, but they may access constants and variables.

Here is the form for a procedure declaration

proc identifier ; compound-statement ;

For example:

```
proc boxofstars;
[  putln("*****");
  putln("      *");
  putln("*****"); ] ;
```

3.7.2 Function.

A function is a subroutine that takes a single parameter and returns a value. It may be called from within expressions.

Here is the form for a function declaration

func identifier (identifier) ; compound-statement ;

The second **identifier** is the name of the temporary parameter variable. This variable must have been previously declared. The result of the function is passed back in the automatically declared variable **return**.

For example:

```
func mult2add3(value);  
[ return = value * 2 + 3 ];
```

Functions may be recursive. That is, they may call themselves directly or indirectly.

3.8 Advanced Statements.

3.8.1 goto Statement.

The goto statement jumps from anywhere within a program to a label.

Here is the form of a label, where integer is a number between 0 and 999,999.

integer :

Here is the form of a goto statement, which jumps to the specified label.

goto integer

Avoid gotos that jump out of or between subroutines. Such jumps skip code that subroutines use to set up and then clean the program stack. Skipping such code will in most circumstances cause the program to crash.

Gotos to a label in the same subroutine or in the main program block are perfectly legal. They can however produce code that is difficult to follow.

Caution: Improper use of **goto** and labels can cause your program to crash and the computer to lock up.

3.8.2 err Statement.

By default when UPL detects a runtime error (such as an attempt to divide by zero) it prints an error message and halts execution of the program. You may however direct UPL to jump to a label within the program to handle the error. Since this is in effect a goto the code you jump to will be responsible for cleaning up whatever has been left on the program stack.

err (integer | **off**)

err off instructs UPL to handle runtime errors in the default manner.

Caution: Improper use of **err** can cause your program to crash and the computer to lock up.

3.8.3 push and pop Statements.

UPL maintains its own data stack separate from the processor stack. The data stack is used internally by UPL to evaluate expressions. You may also access it the push and pop statements, using it as a temporary place to store data.

push expression
pop variable

push pushes an expression on the data stack. **pop** retrieves it. Any elements you **push** on the program stack within a subroutine must be popped before leaving it. You must not attempt to **pop** more values than you pushed.

For example:

```
func fourthpower(number);  
[ push temp;  
  temp = number * number;  
  return = temp * temp;  
  pop temp; ];
```

Caution: Improper use of **push** and **pop** can cause your program to crash and the computer to lock up.

3.8.4 clear Statement.

The clear statement empties the UPL data stack.

clear

The only situation that you may want to use this statement is if you are implementing your own runtime error recover handler.

Caution: Improper use of **clear** can cause your program to crash and the computer to lock up.

3.8.5 mem Array.

mem is an array that gives you access to the the computer's memory. You may assign a value to **mem**, or use it to peek into a memory location from within an expression.

You may assign an integer expression to the memory location **offset + page * 256** with the following form:

mem (offset, page) = expression

You may peek into the memory location **offset + page * 256** with the following expression:

mem (offset-expression, page-expression)

For example:

```
number = mem(250, 0);  
mem(250, 0) = 10;
```

Caution: Improper use of **mem** can cause your program to crash and the computer to lock up.

3.8.6 call Statement.

call constant [**with reg**]
call integer-address [**with reg**]

This calls a machine language subroutine at the specified address. If the keywords **with reg** are specified then the values of the automatically-declared variables **x**, **y**, **a** and **p** are moved into the corresponding 6502 processor registers before the **call**, and their new values replaced afterwards.

Caution: Improper use of these statements can cause your program to crash and the computer to lock up. In particular when using **call with reg** form that you don't accidentally set the 6502 into decimal mode via the **p** register. You can guard against this by forcibly clearing the decimal bit before making the **call with reg**; **p = p and 119**.

3.8.7 mach Statement.

mach ({ integer # , })

The mach statement inserts the specified byte integer expressions directly into the 6502 processor instruction stream. These integers must be between 0 and 255, inclusive.

Caution: Improper use of this statement can cause your program to crash and the computer to lock up.

3.9 Comments.

Comments may appear in a UPL program between adjacent symbols (ie. identifiers, keywords, numbers, character strings and single characters). Comments begin with an opening less-than sign and close with a greater-than sign. They may cover multiple lines. Comments may not be nested.

For example:

```
< This is a comment. >
< And so
    is this!
>
```

Note that UPL does not implement traditional comparison operators such as **<=**. You can however emulate them using bit operators and the **pos** and **neg** functions.

3.10 Example.

3.10.1 Multiplication Table.

```
< This simple example prints a multiplication table. >
var
    factor, index;
[
    putln("Multiplication Table.");
    putln;

    put("Type in a factor: ");
    get(factor#);

    field(4);
    index = 1;
    while not (index = 11) do
    [
        putln(factor#, "*", index#, "=", factor*index#);
        inc index
    ]
].
```

4. Implementation: Commodore VIC-20.

4.1 Package Contents.

See Section 2.1 on Page 9 for a list of Package contents and loading instructions for the VIC-20.

4.2 The Menu.

The menu appears when you start the UPL compiler and editor. You may use the commands on the menu to create, edit and compile UPL programs. You may also save them on tape.

```
Upl compiler/editor
16-Jan-84 v0.1

New          Compile
Read text    Save text
List         Modify
eXecute      Insert
Delete       Object save
Quit
?
```

To invoke a command type the capitalised letter within its name and press enter. For example, to create a new program type “n” for “New”. To execute a program type “x” for “eXecute.”

(An exception to this is the “Insert” command. It expects you to follow the letter “i” with the line number you wish to insert after. For example, “i0” will begin insertion at the beginning of the UPL program. Type “@” on a single line to end insertion.)

4.2.1 New.

You will be asked if you are sure. If you answer ‘y’ then your current program is lost and UPL restated.

4.2.2 Read text.

This loads the saved source code of a UPL program from the tape device. On invoking this command you will be asked if you are sure. If you reply ‘y’ your current program will be lost. You shall then be asked for the filename of the UPL program to load from tape. If you

hit return this will default to “???”. Whatever you type the extension “.upl” will be added. UPL will then search the tape for and load the specified file.

To abort the search press the RUNSTOP key, enter the BASIC command “RUN” and press Return (Enter) to restart UPL.

Emulator Hint: This command will not work on VIC-20 emulators lacking tape emulation (just about all of them). Instead you should use the emulator’s “snapshot” function to restore a saved system state, which will include UPL *and* your UPL program. (See the “Save Text” Command below.)

4.2.3 Save text.

This saves your file on tape. You will be prompted for a filename as described in Section 4.2.2. The UPL program is kept in memory after saving so you may continue working with it.

Hint: It is a good idea to save your UPL program regularly, in case you accidentally lose it or your computer crashes taking your program with you.

Emulator Hint: This command will not work on VIC-20 emulators lacking tape emulation (just about all of them). Instead you should use the emulator’s “snapshot” function to save the system state, which will include UPL *and* your UPL program.

4.2.4 List.

This command lists your program. You will be prompted for the starting line number. If you simply press return this will default to the first line in the program. You will then be asked if you want the lines to be numbered. (Type ‘1’ to number them, or ‘0’ to list them without line numbers.)

Once the listing UPL will pause on each line. Hold the F7 key down to step through the listing. Press the DEL key to halt the listing.

4.2.5 Modify.

This asks you for the number of the line you wish to modify. The line will then appear, and you will be invited to type a new line that shall replace it. You may use the cursor keys to edit up and change it directly, or you may type a new line from scratch. If you decide not to change it enter a line containing only the at-character “@”.

4.2.6 Delete.

You will be asked a range of lines to delete. The lines will be deleted from the first line you specify upto but not including the last line. For example, “4,9” will delete lines 4, 5, 6, 7 and 8. If you ask to delete a large number of lines you will be asked if you are sure; Answer ‘y’ to continue the deletion.

4.2.7 Insert.

The Insert command expects you to follow the menu letter “i” with the line number you wish to insert after. For example, “i0” will begin insertion at the beginning of the UPL program. Type “@” on a single line to end insertion.

For example:

```
Upl compiler/editor
16-Jan-84 v0.1

New          Compile
Read text    Save text
List         Modify
eXecute      Insert
Delete       Object save
Quit

?I0
< An example program. >
[
  putln("Hello world")
].
@

?
```

4.2.8 Quit.

This exits UPL and returns to BASIC. To return to UPL with your program intact type “GOTO 15” and press Return (Enter). This should be done immediately, less you accidentally clear the UPL compiler/editors variables and with them, your UPL program.

4.2.9 Object save.

You may only choose this command after successfully compiling your UPL program. The command saves the object code version of your program (also known as the “executable” or “binary”). This is the runnable version of your UPL program.

You will be prompted to press RECORD and PLAY on your tape device. The object code version of your program will be saved on tape.

You will be asked if you want to verify the save (to make sure there are no media errors). If you want to verify the object code then press STOP and REWIND on your tape to return to where you started saving the object code. Then answer 'y' to verify. You will be prompted to press PLAY on the tape device. The object code will then be verified. You will be alerted if there is an error. If there is try saving again with a different (newer and/or better) tape. If you did not elect to verify the object code then these steps will be skipped.

Emulator Hint: This command will not work on VIC-20 emulators lacking tape emulation (just about all of them). But depending on your emulator you may be able to save the object code directly to disk with the emulator anyway. When UPL finishes compiling it will tell you the location of object code. For example [5009 141 5140] says that the object code is between memory locations 5009 and 5140 inclusive, and is 141 bytes long. (You can run your program by going "SYS 5009" from BASIC). In hexadecimal this memory range covers between \$13EB and \$1414 inclusive, and is \$8D bytes long. If you use your emulator to save a binary file covering these memory locations then you will have saved your program. Additionally the UPL runtime library (which you need to run your compiled program anyway) occupies memory locations \$1009 to \$1390 (4105 to 5008 decimal). If you save the entire memory space from \$1009 to (in this example) \$1414 inclusive then you will have saved your program *and* the runtime library in a single package. You may then load this file on any VIC or VIC emulator and type "SYS 5009" from BASIC to run your program. Could it be any simpler? ☺

4.2.10 eXecute.

You may only use this option after having successfully compiled your program. On invoking this command you will be asked if you are sure. If you reply 'y' then your program shall be run. If there is an error (eg. no object code is present) you will be returned immediately to the menu.

Otherwise your compiled UPL program shall run. It's output shall be surrounded by lines saying "running" and "run ends."

4.2.11 Compile.

The 6502 processor inside the VIC-20 cannot execute UPL directly. We must translate it into 6502 machine code which the processor can run directly. This is the function of the "Compile" command; To take your UPL program and convert it into 6502 machine code. We can then use the "eXecute" command to run the generated 6502 machine code.

When you invoke the command option you'll receive the following prompts. Each prompt shows its default value in brackets. By pressing Return (Enter) without typing anything else you'll take the default value. Otherwise type the value you want and then press Return (Enter),

default options(y)? If you want a normal compilation with all the default values simply hit return. Otherwise 'n'.

above `correct(y)?` Hit return to start the compilation. Otherwise 'n'.

If you did not select the default options you'll have an opportunity to review and change the compilation options:

<code>code start(5009)?</code>	Beginning at what memory location (expressed in decimal) shall the generated 6502 machine code be inserted into memory? The memory area between 5009 and 7679 is guaranteed to be free for your compiled UPL program. If you specify another memory area make sure it is not already being used by something else.
<code>code limit(7679)?</code>	The last memory location in the memory area that may be used to store your compiled UPL program.
<code>stack checking(y)?</code>	The 6502 processor contains a small stack of only 256 bytes. If you're calling a lot of subroutines it's possible to overrun this stack, causing your computer to crash. Selecting this option will insert code that checks the stack during the start of each subroutine call to make sure there's still some room. This additional code will however slow down your program a little.
<code>integers >127(y)?</code>	Normally UPL will only permit integers between -128 and 127. If you answer 'y' to this option it'll also allow integers between 128 and 255. It does this by mapping the range -128 to -1 onto 128 and 255. So for example -1 and 255 are considered to be the same number. (Actually they are by the law of two's compliment. See a computer textbook on binary arithmetic for more details.) If you can handle this concept then answering 'y' to this option will give your UPL program permission to use the full range of byte values in the UPL source. It doesn't disable overflow detection.
<code>auto setup(y)?</code>	Normally when at the start of a compiled UPL program the compiler inserts the following statements; clear; err off; field(0); . These set up the UPL runtime library. If you type 'n' these statements will not be inserted. You should make other arrangements, since without at the very least clearing the UPL data stack your computer may crash.
<code>terminating opcode[b/r](r)?</code>	If you type 'b' when your UPL program finishes execution it'll execute a 6502 BRK (break) statement. This clears the screen and returns to the BASIC command line. If you use 'r' then your program will be terminated with a 6502 RTS (return from subroutine) instruction. This returns it to where you left it; whether it was from a SYS call on the BASIC command line, or a SYS call from within a BASIC program. On returning BASIC will pick up where it left off.

4.3 Compiling.

Here is some sample output from using the compiler:

```
default options(y)?
above correct(y)?
compiling
pass 1
....
pass 2
....
successful compilation
[5009 141 5140]
```

The compiler makes two passes through the source code. The first pass checks for errors and calculates the addresses of the objects in the program. The second pass generates the machine code.¹ A dot is printed everytime a group of lines is processed. This gives you an idea of the relative speed and progress of the compilation.

Those numbers at the bottom tell you the location and size of your program expressed in decimal (base 10). They are in the following form:

[start-address size-bytes last-address]

For example, [5009 141 5140] says that the object code is between memory locations 5009 and 5140 inclusive, and is 141 bytes long. So long as the UPL runtime library is loaded, you can run your program by going “SYS 5009” from BASIC.

4.4 Errors.

If the compiler detects an error during compilation it'll halt and report the error. For example:

```
var
  index;
putln("hello")].      < Note we left the first "[" out >
```

This will report the error thus:

```
compiling
pass 1
...
error 4 in line 3
  putln<<<
  "["
expected.
```

¹ In hindsight, if UPL had been designed so it did not have a **goto** statement and used a different form for the **err** statement then UPL could have been implemented as a single pass compiler. This would have cut compile time in half!

```
"println"  
is not allowed here.  
continue(y)?
```

See Section 4.5.1 on Page 28 to see what compilation error “4” means.

The “`continue(y)?`” is the compiler asking you if it should continue trying to compile the program, even though a fatal error has been found. If you answer ‘y’ then UPL will continue looking for errors. It’ll terminate at the end of the first pass so you may correct the errors and try again.²

4.4.1 ?Out of Memory error.

If the UPL compiler halts with the BASIC error “?Out of Memory error” then your UPL program is too complex for it to parse. This can happen if your expressions are very deeply nested (eg. have too many brackets inside brackets). You can avoid this by splitting such complex expressions up into simpler expressions using simpler variables.

You can return to the menu with your editor intact by typing the following on the BASIC command line:

```
print si: goto 15
```

Do this immediately, otherwise you might enter a command that causes BASIC to lose your UPL program.

4.5 Errors.

In order to preserve memory, the VIC-20 implementation reports errors as numbers. (Having full error descriptions in the compiler/editor would have taken room away for storing your UPL program.) You can look up errors in the following sections.

4.5.1 System Errors.

System errors are generated when you do something that doesn’t make sense. For example, trying to insert a negative line number.

System Error Codes.

0. Impossible request. eg. Inserting a negative line number.
1. No more memory space for source program (150 line limit).
2. Object code save got a verify error.

² Apparently UPL version 0.1 did not implement error recovery. A pity; With UPL’s slow compile time it’d be much better to try and find all the errors in a single parse.

4.5.2 Runtime Errors.

A runtime error is reported when an error is detected while your compiled UPL program is executing. For example, runtime error “4” would be reported with the following message.

```
Error upl 4
```

You can return to the menu with your editor intact by typing the following on the BASIC command line:

```
print si: goto 15
```

Do this immediately, otherwise you might enter a command that causes BASIC to lose your UPL program.

Runtime Error Codes.

0. UPL Data Stack overflow. Too much data has been pushed on the UPL Data Stack. This could be because of a complex expression, not enough space being left to evaluate a simple expression, too many functions being called at once or the **push** statement being overused.
1. Additional or subtraction overflow. An expression has resulted in a integer that exceeds the allowable range of byte values.
2. Division or modulo (remainder) by zero error.
3. Multiplication error. An expression has resulted in a integer that exceeds the allowable range of byte values.
4. Bad field width. The field value must be between 0 and 80.
5. UPL Data Stack underflow. Too much data has been popped off the stack. This can be caused by overusing the **pop** statement.
6. Processor stack overflow. The 6502 processors stack has or is about to overflow. Too many functions or procedures have been called at once.

4.5.3 Compile Errors Codes.

Compile Error Codes.

0. Program is incomplete. Did you forget the terminating dot after the main compound statement? Perhaps the closing “]” on a compound statement? Maybe you forgot a closing double-quote on a character string? etc.
1. Identifier or label declared twice.
2. Too many identifiers or labels declared.
3. This identifier was not declared.
4. Something was expected, but not found. Perhaps the previous statement did not have a terminating semicolon? Check the previous line.
5. Too many structured statements nested inside one another.
6. Label does not appear within the program.
7. Label is expected here.

8. Label of “off” expected here.
9. Machine code integers must be between 0 and 255, inclusive.
10. “inc” or “dec” may only be used on variables.
11. You may only “call” an integer or constant.
12. Only variables may be got with “get”.
13. Not enough memory to store the compiled program. When compiled your UPL program generates too much object code to fit into the allocated memory space. Try simplifying your program, and using common code where possible. If worst comes to worst you can write your program in modules located in different parts of memory and have them call each other. But try simplifying your program first.
14. A constant is not allowed here.
15. Integer exceeds 127. (You may turn this off with the “integers>127” compiler option.)
16. A function is not allowed here.
17. Constant in expression exceeds 127. (You may turn this off with the “integers>127” compiler option.)
18. Factor is expected. A factor is the simplest element in an expression. Factors may be a variable, an integer, a function call, a character string in “put” and “putln”, the “mem”, “pos”, “neg” or “zero” functions, “c” (carry), an expression in (round) brackets or starting with “not”. In other words part of an expression was expected but not found.
19. Only variables may be popped.
20. Integer expected.
21. A function parameter must be a variable.
22. Identifier expected.
23. Integer is too large. Must be 999,999 or less.
24. Identifier not declared.
25. Illegal start of a statement. A statement can’t start with that!
26. Identifier not declared.
27. Illegal separator or statement start. A statement separator “;” or statement should be here, but isn’t.

4.6 Memory Configuration.

You only need to read this if you’re calling machine code other than that created by UPL or poking the memory array.

Do not use the following memory locations (specified in decimal): 251-255 or 673-767 (both used by the UPL runtime library), 4069-5008 (the runtime library itself) or of course the area where the compiled UPL program is stored (the range reported at the end of compilation).

Further note that under UPL a VIC-20 with 16Kb+ of expansion memory is reconfigured to look like an unexpanded VIC; the target system of the UPL system. Screen memory begins at 7,630 decimal and colour memory at 38,400.

4.7 Look Ma! No Compiler!

This section describes how to run your compiled UPL program without needing to load the compiler.

As described above, UPL programs expect to run on an unexpanded VIC. Thus if you are using a 3.5Kb VIC no changes are necessary. However if you are using a VIC with 8Kb or more of expansion memory you must relocate video memory to the above locations. Type:

```
poke 36866, 150: poke 36869, 242: poke 648, 30: print chr$(147)
```

If you still want to load a BASIC program while your UPL program is in memory you'll need to locate BASIC too. This'll depend on the memory area occupied by the compiled UPL program. The following example relocates BASIC to begin at 10,240 decimal (memory page 40). Remember under most circumstances you won't need to do this.

```
poke 43, 1: poke 44, 40
```

Emulator note: Loading and run the BASIC program UPL-BOOT.BAS will run a short machine-language program that moves everything into the right place, including the video memory. After doing this it will try and load the UPL runtime library off tape. Because most emulators don't have tape support you'll need to hit RUNSTOP.

You'll now need to load the runtime library. Simply type the following to load the runtime library off the UPL system package tape.

```
load "uplruntime.bin"
```

Emulator note: With most emulator you can load the runtime library directly into memory straight off disk. Under PCVIC type [Escape][I][U] and enter the path of the runtime library file. Under V20 load the runtime library as if it were an BASIC program. Note that not all emulators can do this, although you could hack a way around it by converting the runtime library so it looks like a ROM cartridge file (albeit one located at \$1000-\$1390, or decimal 4096-5008).

You can now load your compiled UPL program from tape. For example:

```
load "galaxy"
```

When the VIC loads a binary from tape it stores the start address in the memory locations 193 (low byte) and 194 (high byte). Accordingly you should be able to run your program with the following command:

```
sys peek(193)+peek(194)*256
```

Emulator note: Use the emulator to load your binary file directly from disk. Since the VIC won't have set memory locations 193 and 194 for you you'll need to remember the starting location (the entry point) of your compiled VIC program. For example, SYS 5009.

If the computer crashes then maybe you loaded the wrong program, had a tape read error, didn't load the runtime library or relocated video memory incorrectly.

Doing all this setup can be time consuming. To save yourself having to do this every time you want to run the program we recommend you save a single binary file containing the runtime library (4096-5008, \$1000-\$1390) and your compiled UPL program. You can then write a little BASIC program to put on the tape before your saved binary. The BASIC program will write and execute a short machine code program that loads your file and then calls the entry point; usually 5009 decimal.

4.8 Advanced Features.

4.8.1 Suppressing Overflow.

It's easy to move a graphical character across the screen's page boundary by using the carry flag.

```
mem(playerlo, playerhi) = 32; <blank the shape>
playerlo = playerlo + 22;    <move it down a row>
if c                        <if carry is set>
then inc playerhi           <then we need to jump to the next page>
mem(playerlo, playerhi) = 83; <redraw the shape at its new position>
```

Unfortunately UPL will raise an overflow runtime error when `playerlo` exceeds 127. This will happen even when the compiler option allowing numbers > 127 has been set. The compiler option only allows those numbers in the source; It doesn't prevent overflow errors.

In order to disable an overflow you need to add the following code to the start of your UPL program:

```
mem(135,16) = 162;
mem(149,16) = 162
```

You may now execute the above code without any overflow errors. To reenable overflow detection restore those two memory locations to their original values. You must do this before your program exits, otherwise other programs using the Runtime Library will also have overflow detection disabled.

```
mem(135,16) = 112;
mem(149,16) = 112
```

Caution: Placing values other than 162 or 112 in the above memory locations will cause your program to crash or return erroneous results.

4.9 Bugs.

UPL version 0.1 for the VIC-20 contains the following bugs:

- **Integer -128.**

Printing -128 causes a few punctuation symbols to appear instead of “-128”. Also be aware that the negative of -128 under an 8 bit two’s compliment system is in fact -128!

- **Multiplication.**

Multiplications that overflow by a small amount (eg. exceeding 127 by 3) are not detected.

4.10 Example.

4.10.1 VIC-20 Meteor Shield.

```
<meteor shield.
set compiler option integer >= 127 to yes.
compile time 20+ minutes.
block the meteors using l and ;>

cons
    false=0,
    true=-1,
    meteorshape=209,      <A ball>
    shieldshape=214,      <A cross>
    cityshape=177,        <Superstructure>
    blankshape=32,        <A blank>
    homech=19,            <homes cursor when printed>
    clearch=147;          <clears screen when printed>

var score,               <current score>
    shieldlo,            <offset on bottom screen page of shield>
    meteorlo,            <screen page of meteor>
    meteorhi,            <offset on screen page of meteor>
    index,               <initialisation counter>
    loop,                <delay loop counter>
    citydestroyed,       <true when city is destroyed>
    ch,                  <Another game? 'y' or 'n'>
    counter,             <Moves meteor every fifth cycle>
    keytyped;            <current key pressed>

func key;                <Return the current key pressed>
[return=mem(197,0)];

proc delay;              <Waste some time>
[loop=0;
repeat
    inc loop
until loop=150];

func random;             <Return a pseudo-random number>
[call 57492;
return=mem(141,0)!22;
if neg then return=-return];

proc setscreen;          <Clear the screen and draw the city>
[mem(15,144)=8;
put(clearch);
index=228;
repeat
    mem(index,31)=cityshape;
    inc index
until index=250];
```

```

proc setmeteor;           <Set the meteor somewhere on the screen top>
[meteorhi=30;
 meteorlo=random+22;
 mem(meteorlo,meteorhi)=meteorshape];

proc setshield;          <Draw the player shield>
[shieldlo=206;
 mem(shieldlo,31)=shieldshape];

proc movemeteor;          <Move the meteor down one>
[mem(meteorlo,meteorhi)=blankshape;    <Blank the old meteor position>
 meteorlo=meteorlo+22;                <Move it down one row >
 if c then inc meteorhi;              <If carry then go to the next page>

 if mem(meteorlo,meteorhi)=shieldshape    <If meteor hit the shield...>
 then [setmeteor; inc score; put(homech,score#)] <Good! Caught by player!>
 else if mem(meteorlo,meteorhi)=cityshape  <If meteor hit city...>
 then citydestroyed=true                  <City is toast>
 else mem(meteorlo,meteorhi)=meteorshape;  <Hit nothing; draw at new
positon>

proc moveshield;          <Move player shield>
[mem(shieldlo,31)=blankshape;          <Blank old position out>
 keytyped=key;                        <What key did they press?>

 if (keytyped=21) and not(shieldlo=206)    <move left?>
 then dec shieldlo
 else if (keytyped=22) and not (shieldlo=227) <move right?>
 then inc shieldlo;

 mem(shieldlo,31)=shieldshape];          <Draw shield at new postion>

proc title;               <Title screen>
[putln(clearch,142,"Meteor Defense");      <Clear screen>
 putln;putln;putln("type return to play");

 repeat
  get(ch)
 until ch=13]; <Wait for return to be pressed>

proc initialise;          <Initialise for a new game>
[score=0;                  <Reset score>
 counter=0;                <Reset meteor movement counter>

 setscreen;
 setshield;
 setmeteor;

 citydestroyed=false]; <City is ok>

< main program here >
[mem(15,144)=27;          <Set screen colour>
 mem(135,16)=162;        <Patch UPL runtime to ignore overflow>

 title;
 repeat

  < Play a game until the score reaches 100 or the city is destroyed>
  initialise;            <Set everything up>
  repeat

```

```

delay;                <Waste some time>
moveshield;           <Let the player move their shield>

inc counter;          <Move the meteor every fifth cycle>
if counter=4
then
    [counter=0;
     movemeteor]
until (score=100) or citydestroyed;

putln; putln; putln; <Put some space between the score and message >

<Give the player some feedback>
if citydestroyed
then if putln("Oh no! The meteor hit the city!!")
else [putln("You saved the city!");
      putln("You are the wind beneath Wally's wings!");]

putln("play again ?");
get(ch);
put(ch)
until ch="n";

mem(135,16)=112].      <Patch UPL runtime to stop ignoring overflow>

```

5. Improvements.

The UPL Software has been placed in the public domain. You're free to modify it.

But I wouldn't recommend this to anyone. The Compiler/Editor is written in BASIC. Due to speed and memory considerations it contains no comments. I planned the program meticulously on paper, I don't have the time or inclination to scan in those handwritten notes. As a language BASIC is awful, and BASIC programs are practically unmaintainable. *BASIC is basically assembler with a few LEFT\$()-type functions!*

If you're interested in learning how to write compilers I recommend you get a decent book on compiler design, and learn how to write a top-down LL(1) parser. Do it in a high-level language like C or C++. You'll end up with a product that's better, faster and maintainable.

I can't imagine anyone wanting to write software for a VIC-20. Indeed I'd discourage it. The VIC-20's scant memory resources and slow processor forced programmers to write compact and efficient code. With today's bloatware this appears to be a lost art, but there's no practical reason it can't be practised on a modern PC. If you must write a serious application for a VIC-20 (perhaps you're an embedded systems designer who's realised there are a lot of \$5 VICs in garage sales)³ I'd recommend you find a decent 6502 C cross-compiler with optimisation.

In 1983 as a first year computer science student writing the UPL compiler was a terrific learning experience. With a little more time and experience I would have loved to have written a Pascal compiler ROM cartridge that though some cheap and clever connector was able to piggyback with a 16Kb RAM cartridge. The software churned out on the VIC during it's short lifetime was remarkable, but with a decent development system like that it would have been moreso. I've written many compilers since then; My latest being an integrated database scripting language called RedScript that takes the best features of C++, Pascal and Java and rolls them all into one. You can read about RedScript on <http://www.kdef.com>.

Looking back I realise it would have been better to make UPL a 16-bit Pascal or even "C" clone. The byte data types are very difficult to work with. UPL could have used more fundamental data structures like arrays and perhaps structures and enumerated types. The Runtime library should have been smaller too; what's the point in wasting a hundred-or-so bytes on a small system with a number-input routine when the program doesn't need it? Similarly UPL makes no attempt at optimising its output; On a small, slow processor that's a luxury you can ill-afford.

After I finished the VIC-20 implementation of UPL I got and ported it to an Apple II. There it received an Unix-like editor and a substantial performance boost with the Einstein Apple BASIC compiler.

³ I recently picked up a VIC-20 for \$15. All I really wanted was the RF modulator, but I got the VIC-20, a decent Commodore dot matrix printer and 1541 disk drive thrown too!

If I *had* continued with VIC-20 UPL I would have done this...

- Increased the 150 line UPL source limitation. (Easy; It's an arbitrary constant.)
- Added disk drive support. (Very easy).
- Added compiler error recovery, detecting multiple errors per pass. (Not so easy).
- Dropped labels and made UPL a much faster one-pass compiler. (Not that hard.)
- Written a smaller runtime library. (I do have the source for the original on a tape, but I haven't been able to transfer it. If anyone *really* wants it give me a yell.)
- Added peephole and register optimisation. (With a bit of clever coding it could be done.)
- Added simple arrays. (The *mem* function already partially does this.)
- Rewritten the UPL compiler in UPL! (How Zen!)

Following UPL I started writing a full-blown 16-bit Pascal implementation with a friend for the Apple. A week into this new project some guy called Phillippe Kahn came out with a thing called "Turbo Pascal." The rest is history.

6. Emulators.

At the time of writing the following VIC 20 emulators were available. Currently we recommend PCVIC.

Emulator.	Description.
PCVIC 1.14	<p>Author: B.W. van Schooten.</p> <p>Platform: DOS.</p> <p>PCVIC is my favourite emulator, because it's fast! It is a little rough around the edges. For example, to load a BASIC program you must tell PCVIC first to load it, and then to "undelete" it. Fortunately PCVIC supports a snapshot format called "PCV" which saves a lot of this mucking around.</p> <p>wwwhome.cs.utwente.nl/~schooten/software/vic-20/pcvic.html</p>
V20	<p>Author: Lance Ewing.</p> <p>Platform: DOS.</p> <p>Good! Generally easy to use and better file loading interface than PCVIC, but slower. However you'll need to rename any BASIC files <code>.bin</code> in order for V20 to see them. Debug mode could be better done; the seemingly innocuous <code>-b</code> command and not <code>-q</code> will return you to the emulator.</p> <p>crash.ihug.co.nz/~be/vic.htm</p>
Pfau Zeh	<p>Author: Arne Bockholdt.</p> <p>Platform: Linux, Win32.</p> <p>Pfau Zeh is a relatively new arrival on the VIC emulation scene. It has a nice interface, but the Win32 version is very slow on my Pentium 60. Arne recommends you run this on a high-end system with a fast video card.</p> <p>Warning: At the time of writing Pfau Zeh does not include support to load binary programs such as UPL's <code>UPLRTIME.BIN</code>. While you can hack around this it would be much easier to use an emulator that does support binary programs such as PCVIC or V20.</p> <p>stud.fbi.fh-darmstadt.de/~bockhold/pfauzeh.html</p>
VICE 0.14.2	<p>Author: André Fachat and a cast of many.</p> <p>Platform: Unix, DOS, Amiga, others.</p> <p>VICE was originally just a Commodore 64 emulator, but has since had support added for the VIC-20 and PET. One of VICE's nicest features is it's integrated file support; it can simulate a tape or disk drive attached to the emulated machine. Although VICE has been ported to DOS it currently doesn't run under Win95.</p>

	www.tu-chemnitz.de/~fachat/vice/vice.html
VIC-20 1.1	Author: Paul Robson. Platform: DOS. members.aol.com/autismuk/emu.html
VIC-EMU	Author: Pieter van Leuven. Platform: Amiga. ftp.funet.fi/pub/cbm/crossplatform/emulators/Amiga/vicV0.65.lha

Index.

"

"@" Character 24

6

6502..... 32
 BRK..... 26
 Carry..... 12, 32
 Inline machine code..... 20
 Registers 12, 20
 RTS 26

A

asm 20
Assignment statement 13
Auto Setup 26

B

Bugs 32

C

call statement 20
clear statement 19
Code Limit..... 26
Code Start 26
Comments..... 21
Compiling UPL..... 25
Compound statement..... 15
cons 12
Constants 12

D

Data Types 11
 Byte 11
dec statement 13
Delete 23

E

Emulators 38
err statement 18
Error
 ?Out of Memory 28
 Cerror 29
 Compile time 29
 Rerror 29
 Runtime 29
 Serror..... 28
 System 28
Example..... 21, 33
eXecute..... 25
Expressions..... 16

F

field statement..... 14
Files..... 9
func 18
Function..... 17

G

get statement..... 14
goto statement..... 18

I

Identifiers 11
if statement 15
inc statement..... 13
Input 14
Insert 24
 stopping 24
Integers..... 11

L

List 23
Loading 31

M

mach statement 20
mem array..... 19
Memory
 Configuration..... 30
 examining..... 19
 modifying 19
 peeking 19
 poking..... 19
 Reserved areas 30
Menu 22
Modify 23

N

New 22
Numbers
 127 11
 -128 11
 255 11, 26
 Disabling Overflow 32
 Input 14
 Large 26

O

Object save..... 24

P

Package
 Files..... 9
pop statement..... 19
Printing 14
proc 17
Procedure..... 17
Program..... 11
push statement 19
put statement 14
putln statement 14

<i>Q</i>	
Quit	24
Resuming.....	24
<i>R</i>	
Read text.....	22
repeat statement	15
Running	
Without compiler	31
Running compiled UPL.....	25
Runtime	
Reserved Memory	30
Runtime Library	
Disabling Overflow	32
Errors.....	29
<i>S</i>	
Save text	23
Stack	
Checking	26
Processor.....	26
UPL Data.....	26
Subroutines	17
<i>T</i>	
Terminating Opcode	26
<i>V</i>	
var	12
Variables	12
reurn	12
VIC-20 Emulators.....	38
<i>W</i>	
while statement.....	16