

Contents

What is C64List?	2
How to use C64List.....	2
Syntax	2
Converting a BASIC program to text format.....	3
Lines longer than 80 characters	5
Converting from text format back to .prg	6
Supported file types	6
Crunching your program	7
REMark removal.....	9
Text-only comments	10
Line number re-numbering.....	11
Convert to labels	12
Static labels	14
Load Address	15
Custom Tokens	16
BASIC 4.0 and BASIC 7.0 tokenizing	19
Mixed case support.....	19
Graphics glyph support	20
Custom screen codes	21
Including files	21
Conditional compiling	21
C64List Disassembler	22
C64List Assembler	23
Reference Tables	26
C64List syntax.....	29
Future plans	30
C64List versions.....	30

C64List User's Guide

What is C64List?

C64List is a Windows command line based tool to aid in cross-development of Commodore 64 software, particularly BASIC software development. It converts and detokenizes `.prg` and `.p00` files into `.txt` files so they can be edited using your text editor of choice on a Windows machine. And it also tokenizes and converts modified text files back into `.prg` or `.p00` files that can be loaded directly into a C64 machine or simulator such as Vice.

C64List also currently has some limited support for disassembling machine language programs; we intend to add more support for this in the future.

C64List was designed and developed by Jeff Hoag <c64List@gmail.com>.

How to use C64List

The development model that C64List advocates is to convert a C64 BASIC program into text format one time at the beginning of development, making all edits to the text version, and then converting back to `.prg`/`.p00` as many times as necessary. This development model gives you a much a much richer set of features in the text version of the BASIC program than can be supported directly in the binary (tokenized) model, including readable/editable cursor control codes, automatic line number renumbering, and line number-less development utilizing a label model.

The first step to cross development is to devise a way to get `.prg` or `.p00` files from a 1541 disk into the Windows file system. There are numerous ways of doing this including the X1541 cable solution, and over the internet using CommodoreServer.com. We advocate using CommodoreServer with a Comet64 modem.

At this time, C64List supports only the `.p00` and `.prg` file formats; `.d64` support is planned for the future.

Syntax

Just giving the syntax is probably more confusing than just giving an example and explaining it, so here is an example:

```
C64List test.prg -txt -lbl -hex -prg:newfile
```

Of course, C64List is specified first. After that, the first parameter specified without a leading “-” is the name of the file to load. Here, we load “test.prg”. C64List uses the extension of the file to determine what format to expect, so in this example, it knows we are loading a tokenized, BASIC program file.

The second and following parameters tell C64List what formats to output. -txt tells C64List to convert the loaded file into text format and save it as “test.txt” – the name of the loaded file is used for the target filename, appended with the specified extension. In our example, we requested C64List to output 4 files in 4 different formats:

- “test.txt” (a text file)
- “test.lbl” (a differently formatted text file)
- “test.hex” (a hex-dump format), and
- “newfile.prg” (a tokenized BASIC file)

Notice that we told C64List to name the output .prg file with a different name to avoid stomping on our original input file. The new filename is specified by typing the new file after the format specifier, separated by a colon. We could have given all four output files completely different names if we had wanted to. Finally, you can also give an output file a different extension if you prefer. Simply use the full name and extension of the filename you want to use.

If you tell C64List to output to a filename that already exists, it will complain and fail to output the file. You can override this behavior using the -ovr parameter to “force” C64List to overwrite the file.

In a single command line, C64List will output up to 1 of each file format that it supports.

If you need help remembering what parameters are available for use, just type

```
C64List -h
```

And it will print out a little help screen. Note that the -h is optional—including no parameters has the same effect.

We will see more command line examples in the following pages.

Converting a BASIC program to text format

The following examples use a file named test.prg, a working tokenized BASIC program. Converting test.prg file into a text-formatted file is done with the following command line:

```
C64List test.prg -txt -crsr
```

This will read test.prg, detokenize it and save the result as test.txt. Looking at the resulting file, you will see it looks pretty much like what you’d get from typing LIST on a C64 machine:

```

100 REM THIS IS A BASIC PROGRAM FOR TESTING C64LIST
110 GOSUB10000
120 PRINT"THIS IS A BASIC PROGRAM"
130 PRINT"FOR TESTING C64LIST"
140 PRINT
150 GOSUB2000
160 PRINT"-----"
170 GOSUB230
180 PRINT
190 PRINT"-----"
200 PRINT:PRINT:PRINT
210 PRINT"  _  ~  >  æ'"
220 END
230 RESTORE
240 READA$
250 IFA$="*"THEN GOTO290
260 PRINTA$;" ";
270 GOSUB1000
280 GOTO240
290 RETURN
1000 REM A SHORT DELAY LOOP
1010 FORDE=0TO100:NEXT
1020 RETURN
2000 REM A LONGER DELAY LOOP
2010 FORDE=0TO1000:NEXT
2020 RETURN
10000 POKE53280,6
10010 POKE53281,0
10020 RETURN20000
20000 DATAC64LIST,CAN,DE-TOKENIZE,THIS,PROGRAM,AND,CONVERT,IT,INTO,A,TEXT,FILE,
20010 DATASO,YOU,CAN,EDIT,IT,IN,WINDOWS,USING,THE,TEXT,EDITOR,OF,YOUR,CHOICE,*

```

The main problem with this listing is that the cursor control codes are essentially unintelligible and uneditable: since the C64 does not use ASCII, the codes do not appear like they would on a C64, and they are difficult if not impossible to produce on a text editor in Windows. Therefore, C64List has a special function to convert these codes to something usable. This is actually the default operation; to illustrate this function in the above example, we suppressed that feature by specifying `-crsr`.

Note: if you specify `-crsr`, although the graphics characters will appear as unintelligible ASCII characters, if left alone, will convert back without issue.

Once we remove the `-crsr` option and try again with the following command line, we get something more readable. Also note the `-ovr` parameter was also added; it tells C64List to overwrite the previously-created test.txt file for us—otherwise C64List would complain and not output our new file.

```
C64List test.prg -txt -ovr
```

```

100 REM THIS IS A BASIC PROGRAM FOR TESTING C64LIST
110 GOSUB10000
120 PRINT"{clear}{gray1}{down:5}THIS IS A {red}BASIC{gray1} PROGRAM"
130 PRINT"FOR TESTING {blue}C64LIST{gray1}"
140 PRINT
150 GOSUB2000
160 PRINT"-----"
170 GOSUB230
180 PRINT
190 PRINT"-----"
200 PRINT:PRINT:PRINT
210 PRINT"{rvrs on}{black} {gray1} {gray2} {gray3} {white} {purple}{rvrs off}"
220 END
230 RESTORE
240 READA$
250 IFA$="*"THEN GOTO290
260 PRINTA$;" ";
270 GOSUB1000
280 GOTO240
290 RETURN
1000 REM A SHORT DELAY LOOP
1010 FORDE=0TO100:NEXT
1020 RETURN
2000 REM A LONGER DELAY LOOP
2010 FORDE=0TO1000:NEXT
2020 RETURN
10000 POKE53280,6
10010 POKE53281,0
10020 RETURN
20000 DATA C64LIST,CAN,DE-TOKENIZE,THIS,PROGRAM,AND,CONVERT,IT,INTO,A,TEXT,FILE
20010 DATA SO, YOU,CAN,EDIT,IT,IN,WINDOWS,USING,THE,TEXT,EDITOR,OF,YOUR,CHOICE,*

```

The user may now edit the text file with the BASIC code in it if desired.

You can see in this listing that C64List replaced the binary codes with handy, editable cursor control codes, such as {clear} and {blue}. Additionally, when C64List found a consecutive, repeated cursor code (see “down” in line 120), instead of outputting “{down}{down}{down}{down}{down}”, it output a much more manageable “{down:5}”. The list of all available cursor codes appears in the reference sections below.

Lines longer than 80 characters

Using C64List, you may ignore the 80 character length limit for a single program line. This is a limitation of the C64’s BASIC editor, not the language or parser itself, and a .prg file containing lines > 80 characters will run fine.

The caveat is that you will not be able to modify these long lines C64 itself, so if you intend to make any modifications to your software on the C64, you should be cautious about using long lines. On the other hand, if you will make all your modifications using C64List, then go ahead and make them longer than 80 characters—they will take up less memory that way!

Converting from text format back to .prg

To convert the example program back to a .prg file, use the following command line:

```
C64List test.txt -prg:test2
```

If we would have specified `-prg` without `:test2`, C64List would attempt to use the input file name for output, and want to overwrite our original `test.prg` file. However, since `-ovr` was not specified, it would find the original `test.prg` file and complain and exit without writing anything.

The `-prg:test2` syntax tells C64List to create a file named `test2.prg` instead of using the same name as the input file. Furthermore, if the user wishes, C64List allows the extension to be specified as well; for example `-prg:test2.bin`.

Now, if `test.txt` was not edited between when it was created then converted back to a .prg file, the doubly-converted file (`test2.prg`) will be identical to the original file (`test.prg`).

If you are typing in a program using the text format, you may type your BASIC keywords and variables in lower case characters in the text formatted program, and C64List will automatically convert them to uppercase as appropriate. (Keywords and variables will always be output in uppercase when converting to text.)

Supported file types

The first command line parameter on the C64List command line tells what C64List what file to read, and what file format to expect. Since the extension of the input file determines the file-type, the file's extension must correctly identify in which format the file is stored. C64List will *read* the following file types:

.prg	A tokenized BASIC, or machine language program.
.p00	A popular 8.3-compatible file format.
.txt	A text file, as output by C64List.
.lbl	Mostly interchangeable with .txt

C64List can write one or more of the output file formats it supports in a single pass. To output a specific file format, add its option to the command line as follows. Each output file supports the naming option `:<filename>.<ext>` as explained above. C64List will *write* the following file types:

-prg	A tokenized BASIC, or machine language program.
-p00	A popular 8.3-compatible file format.
-hex	An 8-bit hex dump of the program, as if it were loaded into C64 memory.
-asm	A disassembled listing of a machine language program (currently very limited!)
-txt	A detokenized, text version of a BASIC program, with various options.
-lbl	Same as .txt, except with line numbers removed (see explanation below).

The .prg format is an exact binary image of a Commodore 64 file as stored on a 1541 disk. It contains a

load address followed by consecutive bytes of data that are to be stored at and following the load address. If the file is BASIC, the data is encoded in C64 tokenized form, and is, for practical intents, not human readable. If the file is machine language, the data simply contains the machine codes as it was programmed, and is completely not human readable. This file format is supported by many PC-based C64 emulators, including VICE, and by CommodoreServer.com, and is the preferred binary format for C64List.

The .p00 format is similar to the .prg format, but it includes a header containing information about the 1541 directory entry in the file data. This is an old format which was intended to get around DOS's 8-character filename limit, since the Commodore 64 supported longer filenames than DOS allowed. Since Windows supports long filenames, this format is much less necessary these days. C64List only supports files of this form if they have the .p00 extension; .p01 through .p99 are not supported. This file format is supported by Vice and some other emulators.

The .hex format is a text file that shows a formatted hex-dump of memory starting at the load address and continuing for the length of the file. It is intended to show what the program would look like if it were loaded into the Commodore 64's memory and a hex dump were done on the C64 itself. C64List will write this format, but does not currently support loading files in this format.

The .asm format disassembles a (presumably machine language) program, starting at the load address by default. As with any disassembler, if there is inline data, disassembly listing may become garbled. Further enhancements are planned that will make this feature easier to use.

The .txt format is an ASCII-based text file that contains a BASIC program in its untokenized form and is the most flexible format for C64List use. It is intended to be the format of choice for editing BASIC programs that are to be later converted back to a tokenized, binary form. The exact format of this file depends on a number of C64List options.

The .lbl format is a detokenized text format without line numbers. See the following sections for more information on how to use this format. This file type is intended to be transitory, and is for use when initially converting a tokenized BASIC program into a text-based project. The user should use this option to convert the program to text, and then should edit it and rename it to a .txt file before converting it back to a tokenized form. When loading a .txt file, C64List will automatically identify numbered or numberless format and load it as needed.

Crunching your program

Since the RAM in a Commodore 64 computer is rather limited, at times you may need to employ some tricks to save memory. When you type a BASIC program into the C64, it stores all the spaces you type, along with your code. These spaces make your code more readable, but every space stored in memory chews up one byte. This can add up quickly. If your program is too big to fit into memory, one thing you can do is to remove all the unnecessary space characters in your program.

C64List will do this for you if you ask it to, either by specifying the `-crunch` parameter on the command line, or by giving the `{crunch:on}` directive in your text-formatted file. When crunching is enabled, and C64List converts from a text-type file to a binary-type file (tokenizing), C64List will automatically remove

all unnecessary spaces in your program. Spaces inside quotes are left alone, since they are necessary to make your program output look right.

If, for some reason you would like to turn crunch mode off somewhere in your program, you may also use the **{crunch:off}** directive.

Going back to our previous code example, you may notice that it is difficult to read, because the original author “crunched” the BASIC code by removing all the unnecessary spaces. C64List can make your life much easier with its Uncrunch function when you are converting a tokenized file to text format. Uncrunch strategically inserts spaces into the output file to make it much more readable. Specifying the `-crunch` option when converting from a binary format file to a text format file (detokenizing) will activate uncrunch mode. The example in the next section shows the results of using the uncrunch function.

REMark removal

Good programmers use lots of comment in their code. However, due to the limited amount of space in the C64, you may need to remove all your nice REMs in order to squeeze your program into available memory. C64List will do this for you as well. When `-rem` is specified on the command line during a conversion to a binary format, C64List will remove all REM statements and the following remarks before storing the file. C64List also checks to see if the REM is on a line with other code. If so, it will also strip the colon before the REM statement. And if the only thing on a line is a REM statement, C64List will completely remove the line from the program. One nice thing about using C64List to develop your programs is that you can put all the comments you want into the text file, and C64List will handily remove them when converting your program to .prg format, but the comments will remain safely in your original text file.

You can turn on and off the REMark removal feature right inside your text formatted BASIC file, using the **{remremoval:on}** and **{remremoval:off}** directives. This is useful if you wish to preserve some, but not all of the REMarks in your program.

Specifying `-rem` when converting from a binary to text format has no effect.

Normally, it would be dangerous to simply strip all REM statements from a BASIC program; if a GOTO or GOSUB targets a line that contains nothing but a REMark statement, the line is completely removed. This means that an `?UNDEF'D STATEMENT` error would occur when you attempt to run the program later. However, C64List is smart enough to automatically change any GOTO or GOSUB statements that point to a removed line, so that it targets the next line that actually contains BASIC code after the point where the line was removed.

If either the REMark removal or crunch options is used, the re-tokenized binary file is no longer guaranteed to be identical to the original binary file, even if no editing was done to the intermediate text version. This is because these options cause C64List to automatically edit the file for you.

Here is the same example above, with the `-crunch` option:

```
C64List test.txt -txt:test2 -crunch
```

```

100 REM THIS IS A BASIC PROGRAM FOR TESTING C64LIST
110 GOSUB 10000
120 PRINT"{clear}{gray1}{down:5}THIS IS A {red}BASIC{gray1} PROGRAM"
130 PRINT"FOR TESTING {blue}C64LIST{gray1}"
140 PRINT
150 GOSUB 2000
160 PRINT"-----"
170 GOSUB 230
180 PRINT
190 PRINT"-----"
200 PRINT:PRINT
210 PRINT"{rvrs on}{black} {gray1} {gray2} {gray3} {white} {purple}{rvrs off}"
220 END
230 RESTORE
240 READ A$
250 IF A$="*"THEN GOTO 290
260 PRINT A$;" ";
270 GOSUB 1000
280 GOTO 240
290 RETURN
1000 REM A SHORT DELAY LOOP
1010 FOR DE=0 TO 100:NEXT
1020 RETURN
2000 REM A LONGER DELAY LOOP
2010 FOR DE=0 TO 1000:NEXT
2020 RETURN
10000 POKE 53280,6
10010 POKE 53281,0
10020 RETURN
20000 DATA C64LIST,CAN,DE-
TOKENIZE,THIS,PROGRAM,AND,CONVERT,IT,INTO,A,TEXT,FILE,SO,YOU,CAN,EDIT
20010 DATA IT,IN,WINDOWS,USING,THE,TEXT,EDITOR,OF,YOUR,CHOICE,*

```

Notice how C64List added some nice spacing into the listing so it is much easier to read.

If you convert from a .txt or .lbl file directly to a .txt file, crunch mode may be much less obvious in the output file, since both the crunch and uncrunch functions will be activated and deactivated at the same time.

Text-only comments

C64List supports a special type of comment that is not native to the C64 as well. When you are developing a program using the text format, you may put as many of these comments as you wish, and C64List will strip them out automatically for you. A single quote mark (') begins one of these comments, and it continues to the end of the line. These comments may be placed at the end of a line of code, or stand-alone on their own line as shown in the following example:

```

'this comment will be completely removed
REM THIS IS A NORMAL REMARK THAT CAN REMAIN IN THE PROGRAM
110 GOSUB 10000 'this comment will also be removed
120 PRINT "THIS HERE ' IS NOT A COMMENT"
130 DATA THIS ' IS NOT A COMMENT EITHER
140 REM NOR IS THIS ' ONE

```

Tick comments cannot be placed inside quotes (since tick is a legal character to print), nor on a line after a DATA statement or REM statement (since a tick is both a valid DATA and REM character).

Line number re-numbering

C64List allows you to completely re-number a BASIC program. To do this, you must first convert the program to text format. There is no mechanism to renumber directly from a tokenized binary file. Once in text format, edit the file and add renumbering directives into the text, and then convert it back to tokenized BASIC. The renumbering options are quite flexible, and allow you to number different parts of the program differently. All GOSUB/GOTO targets are automatically updated to point to the new line numbers. The re-tokenized file will be different from the original binary file (and it may even be a different length) due to the line number changes, but the resulting program will execute in the same manner as the original.

To renumber a program, the only thing necessary is to add the directive **{renumber}** at the top of the file. This will cause the whole file to be renumbered from the beginning, starting with line number 0 and incrementing by one for each line.

However, if finer control over the resulting line numbers is desired, other directives may be added at any point in the program. It is advisable, but not imperative, to place these directives on lines that do not contain BASIC code. The renumbering directives are described below:

{renumber}	Request that the following BASIC program be renumbered. Restriction: must be placed before any BASIC code in the file.
{number:<line number>}	Causes the next line of BASIC code to have the specified line number. Restrictions: <ul style="list-style-type: none">• The requested line number must be greater than any line number encountered so far in the file; if this requirement is not met, the directive will be ignored.• The specified value must be a valid line number for the C64 (0-63999)
{step:<value>}	Causes the line numbering to increment by the specified value. Activates on the <i>second</i> line of BASIC code after the directive. Restrictions: <ul style="list-style-type: none">• Must be positive• Must be small enough to allow all lines in the program to be assigned valid line numbers
{nice:<value>}	Causes the next line number to be “niced” to a multiple of the given value. Typically, the specified value should be some factor of ten. For example, if the next line number to be assigned is going to be 437, specifying {nice:100} will instead cause the next line number to be 500. Restriction: Must cause the next line number to be within the valid range.

The following example shows how to renumber a BASIC program. Once the program has been converted to text, add the lines with the renumbering directives as shown:

```

{renumber}
27 X=99
28 GOTO 39
{nice:100}{step:100}
29 X=100
39 PRINT"{down:2}THIS IS LINE 39"
42 PRINT"THIS IS LINE 42"
48 GOSUB 93
67 IF X=99 THEN GOTO 29
68 END
{nice:1000}
{step:10}
93 FOR D=0 TO 1000
94 NEXT
95 RETURN

```

Next time you convert this file, it will magically be renumbered as you requested!

```
C64List renum0.txt -txt:renum1
```

Gives us the following listing:

```

0 X=99
1 GOTO 200
100 X=100
200 PRINT"{down:2}THIS IS LINE 39"
300 PRINT"THIS IS LINE 42"
400 GOSUB 1000
500 IF X=99 THEN GOTO 100
600 END
1000 FOR D=0 TO 1000
1010 NEXT
1020 RETURN

```

Notice how lines 27 and 28 were renumbered to 0 and 1, since the only directive received to this point was a renumber. After we gave the {nice:100} and {step:100} directives, the numbering started at 100 and increased by 100 thereafter, until we got to the {nice:1000} and {step:10} directives. Then the numbering started at 1000 and incremented by 10.

Convert to labels

Anyone who has ever programmed in Commodore BASIC has undoubtedly run into issues with line numbering. For example: didn't leave enough space between line numbers, or decided to move a block of code earlier or later in the program. Although the renumber function described above will help fix these issues, a more convenient method is to eschew line numbers entirely. C64List allows you to do this. Of course, the Commodore would not be happy with a BASIC program that did not have line numbers, so line numbers need to be added back in at some point. C64List takes care of this for you automatically. In the meantime, while working with the numberless code, one needs to be able to define GOTO and GOSUB target points in the code. C64List allows the developer to identify such points using labels, and reference them in the control transfer statements. Like all other C64List directives, labels appear within curly braces. They are identified as labels by a colon immediately following the opening brace. No spaces are allowed between the opening brace and the colon.

If you have an existing BASIC program and would like to convert it to use labels instead of line numbers, C64List will convert it for you. Just specify the output type -lbl. This will output a text file just like normal, except that all line numbers are removed. In their place, where necessary, labels are inserted. Only lines that are targets of a GOTO, or GOSUB, etc. will contain a label, and all references to these labels will be converted to the name of the given label. Labels are automatically assigned by creating a label that is derived from the original line number of the target line. For example, take the following nonsense program:

```
10 PRINT"THIS IS THE FIRST LINE OF THE PROGRAM"
11 REM REM-ONLY LINE WITHOUT ANY REFERENCES
20 REM REM-ONLY LINE WITH A REFERENCE
30 GOTO 50
40 THIS LINE WILL CAUSE A SYNTAX ERROR IF EXECUTED
50 PRINT"MADE IT SUCCESSFULLY PAST THE ERROR"
60 END
70 GOTO 20:REM JUMPING TO A REM STATEMENT
```

Run it through this C64List command line:

```
C64List labeltest.txt -lbl
```

And it will be converted to this:

```
PRINT"THIS IS THE FIRST LINE OF THE PROGRAM"
REM REM-ONLY LINE WITHOUT ANY REFERENCES
{:20}
REM REM-ONLY LINE WITH A REFERENCE
GOTO {:50}
THIS LINE WILL CAUSE A SYNTAX ERROR IF EXECUTED
{:50}
PRINT"MADE IT SUCCESSFULLY PAST THE ERROR"
END
GOTO {:20}:REM JUMPING TO A REM STATEMENT
```

Notice that there are no line numbers, and that two labels have been inserted that look surprisingly like line numbers that used to be in the program. C64List found that these two lines of BASIC code targets of a GOTO or GOSUB, so it automatically inserted the labels, and changed the GOTO statements to specify the labels rather than the line numbers. Most likely you will want to use your text editor to search and replace all instances of each label with a more meaningful word. Perhaps in another 30 years C64List will be smart enough to insert meaningful labels for you using its super-artificial-intelligence code understanding engine, but for now it leaves that task to you. Regardless of the actual labels (C64List-generated, or text selected by you), C64 will be able to convert the program back into line-numbered code.

Earlier we talked about how removing all REM statements could be dangerous due to the fact that some GOTOs or GOSUBS might call a line that only contains a REM statement. C64List is just as smart about removing REM statements in the labeled format as it is the line-numbered format and using the -rem option is completely safe. If a control transfer statement's target is a line that was completely obliterated due to only containing a REM statement, the label will be preserved and any control transfers to the line will be saved. Adding a -rem to our above example

```
C64List labeltest.txt -lbl -rem
```

Will convert it to this:

```
        PRINT"THIS IS THE FIRST LINE OF THE PROGRAM"
{:20}
    GOTO {:50}
    THIS LINE WILL CAUSE A SYNTAX ERROR IF EXECUTED
{:50}
    PRINT"MADE IT SUCCESSFULLY PAST THE ERROR"
    END
    GOTO {:20}
```

Now there are no REM statements at all. Yet now, the GOTO 20—which used to point to a line with only a REM statement—now has a label for a target, and so is still valid. Creating a .txt file (as opposed to a .lbl file) with the `-rem` option on this program would create a faulty program.

Now that we have a nice, labeled BASIC program, we can edit it and change the labels to something meaningful. Also, don't forget to set the renumbering options if you wish. Here we have manually edited our example to use labels that actually mean something:

```
{renumber}
    PRINT"THIS IS THE FIRST LINE OF THE PROGRAM"
{:FormerRemOnlyLine}
    GOTO {:SkipErrorLine}
    THIS LINE WILL CAUSE A SYNTAX ERROR IF EXECUTED
{:SkipErrorLine}
    PRINT"MADE IT SUCCESSFULLY PAST THE ERROR"
    END
    GOTO {:FormerRemOnlyLine}
```

Now when we convert it back to .txt format, we get a BASIC program that has been renumbered as per our instructions, even though there were no line numbers at all specified in the source file! We could have been more specific with our line numbering instructions, but in this case we didn't care.

```
C64List editedtest.txt -txt
```

```
0 PRINT"THIS IS THE FIRST LINE OF THE PROGRAM"
1 GOTO 3
2 THIS LINE WILL CAUSE A SYNTAX ERROR IF EXECUTED
3 PRINT"MADE IT SUCCESSFULLY PAST THE ERROR"
4 END
5 GOTO 1
```

Static labels

If you want to excise a block of BASIC code, but you want C64List to act like it is still there, you can use the `{assign}` directive. This is useful for some programming models where there is a resident set of BASIC code at a certain range of line numbers, and dynamic blocks of BASIC code that can be swapped in and out at will.

For example, if you have some resident BASIC code that is numbered from line numbers 0 to 999, and then your dynamic code blocks start after that, you can write each dynamic code block independently from the resident part.

The {assign:<label>=<line number>} tells C64List that whenever it finds a reference to <label>, to treat it as if line number <line number> exists, even though it doesn't. Here is an example:

```
{assign:DrawGrid=100}
{assign:ClearGrid=120}

GOSUB{:ClearGrid}
GOSUB{:DrawGrid}
END
```

C64List can create a valid BASIC program from this code. Of course it will not run properly without a system to link in the resident code.

The {assign} directive does not reserve any line numbers, or modify renumbering in any way. Therefore it is up to the user to prevent these line numbers from being assigned during renumbering.

You can also use this feature during development if you want to suppress errors due to missing lines before you are done with the development.

Load Address

If, for some reason, your program needs to load to an address other than the default address 2041 (\$0801), you can tell C64List to put the code wherever you want in memory. Add the {loadaddr:<address>} directive to the top of your text-formatted BASIC program, and when C64List converts it to a tokenized program file, it will store it at the address you requested. For example:

```
10 PRINT "HELLO C64 USERS!"
20 END
```

Converting to .hex format will give us the following:

```
C64List addrtest.txt -hex
```

```
0800 || 1a 08 0a 00 99 20 22 48 45 4c 4c 4f 20 43 36 ..... "HELLO C6
0810 || 34 20 55 53 45 52 53 21 22 00 20 08 14 00 80 00 4 USERS!". ....
0820 || 00 00 ..
```

As you can see from the address given on the first line of the .hex file, the program is located at the standard C64 BASIC load address, \$0801. However, if we add the following directive to the top of our program, we can see how the address changes.

```
{loadaddr:20481}
10 PRINT "HELLO C64 USERS!"
20 END
```

```
C64List addrtest.txt -hex
```

```
5000 || 1f 50 0a 00 99 20 22 48 45 4c 4c 4f 20 43 36 .P... "HELLO C6
5010 || 34 20 55 53 45 52 53 21 22 00 25 50 14 00 80 00 4 USERS!".%P....
5020 || 00 00 ..
```

And voilà! We see that we successfully changed where the program loads to 20481 (\$5000). Notice that

the line links are also set correctly. This can be useful for editing programs that were written for the PET instead of the Commodore 64, for example, since the PET's standard BASIC load address is different—1024 (\$0400).

Custom Tokens

The BASIC keyword list that C64List recognizes can be expanded for the case when you are developing BASIC software using a BASIC extension package. These packages often add new BASIC keywords that are not part of the regular C64 BASIC language. C64List allows you to develop software for these extensions by changing its Tokenizer engine.

The most common usage for this feature is to convert an extended BASIC program in text format to the tokenized format. To use this feature in this manner, add the {tokenizer} directive to the top of your BASIC-formatted text file. The syntax of the Tokenizer engine settings is `<token value>="<keyword>"`. You may add any number of token replacements, separated by commas or semicolons. The token values must be between \$80 and \$FF, and may be specified in either decimal, or hexadecimal using the standard Commodore style \$ prefix. Spaces are allowed, but not required, between definitions. Here is a nonsense Tokenizer modification with the proper syntax:

```
{tokenizer:$cc="newtoken1",$cd="newtoken2"; 206="newtoken3"}
```

It is possible to add token numbers that are not defined in C64 BASIC, and also to replace token numbers that are already defined in C64 BASIC, although the latter is not recommended. C64 BASIC uses all the tokens in the range \$80 through \$CB, and also \$FF. This leaves \$CC through \$FE as undefined.

Please note that C64List can do very little checking on Tokenizer modifications, and it is quite possible to confuse the Tokenizer if you set this up incorrectly. For best results,

- Closely check your spelling and token numbers
- Place the {tokenizer} directive before any BASIC code
- List tokens in numerical order
- Avoid replacing existing tokens, if possible

Token ordering: Newly defined tokens are stored in the tokenizer in the same order they are defined, except for tokens that replace other tokens, which are stored in the former token's location. Correct tokenization depends on the order of the tokens in the tokenizer, due to the way C64BASIC ROMs work. For example, the keyword INPUT can be found in the keyword INPUT#. The C64 can correctly tokenize each of these keywords, however, because the keyword INPUT# is checked before the keyword INPUT. If the order were incorrect, INPUT# might be interpreted as the keyword INPUT, followed by a token for the number sign. This would, of course cause a syntax error, and not operate as one would expect. Your customized tokens must follow the same strategy, and take into account the already-defined tokens as well.

New tokens may be specified in either upper or lower case; they will automatically be converted to uppercase, and treated just like C64List treats any other BASIC keyword.

C64List will report each keyword addition or replacement it finds as it is loading the text-formatted BASIC file.

As of release 2.22, you may now use custom tokens when converting from .prg format to .txt. To do this, create a file containing only the {tokenizer} directives, as explained above, and add a command line parameter to load the tokenizer:

```
-tokenizer:<tokenizer file>
```

This will cause C64List to load your custom tokens before the .prg file is loaded so it will know how to de-tokenize the code.

Starting with release 2.30, if you attempt to convert a .prg/.p00 file that has tokens that C64List does not understand, and you have not provided a tokenizer file to help out, C64List will simply output the hexadecimal value of the token enclosed in curly braces. C64List can also tokenize this format back into a .prg/.p00 file. Here is an example that shows how C64List deals with custom tokens and unknown tokens.

Given the following program CustomTokens.txt:

```
{tokenizer:$e0="GRAB",$e1="EAT",$e2="DROP"}
{tokenizer:$e3="SLEEP"}
10 for i=0 to 100
20 grab "A BURGER AND A COKE"
30 eat "BURGER":drop "COKE"
40 sleep:next
50 end
```

We'll convert it to a .prg file that some weird extension of C64 BASIC would be able to run:

```
C64List CustomTokens.txt -prg
```

We see that C64List tokenizes the program without any complaints, even though it has weird, nonstandard BASIC keywords in it. If you attempt to load this .prg file into a C64 without the associated BASIC extension, you will see that it doesn't make much sense, and it won't run. Now let's take this .prg file and convert it back to a text file:

```
C64List CustomTokens.prg -txt:BadTokens
```

We'll get a file named BadTokens.txt that looks like this:

```
10 FOR I=0 TO 100
20 {$e0} "A BURGER AND A COKE"
30 {$e1} "BURGER":{$e2} "COKE"
40 {$e3}:NEXT
50 END
```

You can see here, that as C64List was detokenizing the .prg file, it ran into the nonstandard tokens and realized they were garbage, so it simply output the bad tokens in a raw token numerical form. This is new as of release 2.30. C64List can't know what these nonstandard tokens mean when converting from the .prg format, since there is no way for the file to indicate what the tokens mean—it is simply a normal C64 formatted program file and does not have the tokenizer data like our original .txt file has. Incidentally, if you attempt to convert this text file back to a .prg file, C64List will happily do just that, and you will get an exact replica of TokenTest.prg. You may also insert tokens by hand in this format if you have the need to. C64List finds the numerical token, converts it to an actual token of the same value, and inserts it into the program. Note that you may use either { \$xx } hexadecimal or { nnn } decimal notation if you do this by hand. C64List will always output the hexadecimal format.

Since we need to tell C64List how to interpret these tokens, we'll create a Tokenizer file, as indicated above. The Tokenizer file is formatted exactly the same as the first few lines of our original text-formatted BASIC file. We'll call it BurgerToks.txt:

```
{tokenizer:$e0="GRAB", $e1="EAT", $e2="DROP"}
{tokenizer:$e3="SLEEP"}
```

Now we will attempt to convert our .prg file to text again, but this time we can use our new file to tell C64List what our strange tokens mean. We'll also add the -verbose parameter so we can see what C64List is doing:

```
C64List
CustomTokens.prg -tokenizer:BurgerToks.txt -txt:GoodTokens -verbose
```

Now when we run, we get the following output in the file GoodTokens.txt:

```
-----
                        C64List (v2.30) Copyright 2011 by Jeff Hoag
-----
Loading tokenizer file: BurgerToks.txt
Modifying BASIC tokenizer
Adding token $e0 = GRAB
Adding token $e1 = EAT
Adding token $e2 = DROP
Modifying BASIC tokenizer
Adding token $e3 = SLEEP
Reading file CustomTokens.prg
Loading C64 BASIC file from prg formatted file: CustomTokens.prg
Load successful
Saving in txt format: GoodTokens.txt
Encountered: 0 Errors; 0 Warnings
C64List finished.
```

Here, we can see that C64List found our tokenizer file and loaded the custom tokens, and then it went on to load the .prg file. When we open GoodTokens.txt, we see that the nonstandard tokens were converted back to text correctly this time:

```
10 FOR I=0 TO 100
20 GRAB "A BURGER AND A COKE"
30 EAT "BURGER":DROP "COKE"
40 SLEEP:NEXT
```

50 END

Finally, as of release 2.30, you may undefine existing tokens. In the {tokenizer} directive, simply specify the token you wish to undefine, and set it to a blank string:

```
{tokenizer:$80=" "}
```

Normally, you won't want to remove tokens from the tokenizer, since it defeats the purpose of C64List's tokenizing feature! It may occasionally come in handy for some custom token schemes, however.

BASIC 4.0 and BASIC 7.0 tokenizing

Although C64List, as its name suggests, is specifically for the Commodore 64, some support has been added for BASIC 4.0 and BASIC 7.0 for users that like to develop for those versions of BASIC. To enable the BASIC 4.0 token map, add the `-b4` command line parameter. Likewise, to enable the BASIC 7.0 BASIC token map, use the `-b7` parameter.

Please note that C64List only supports the single-byte tokens in BASIC 7.0, and not the extended tokens. Therefore, the two prefix tokens, `$CE` and `$FE` are left undefined. This should be a rather small inconvenience however, because it appears that the extended tokens are primarily for use in immediate mode, and not to be put in programs.

The `-b7` and `-b4` parameters are handled in exactly the same manner as other custom tokens: two files have been provided that define these custom token maps. The `-b4` parameter is equivalent to invoking `-tokenizer:BASIC4_0.txt`, and `-b7` is equivalent to invoking `-tokenizer:BASIC7_0.txt`. The shortcuts were provided to minimize the number of keystrokes necessary for these functions.

Mixed case support

The C64 has two built-in character sets. First is the normally-used character set where all alpha characters are uppercase, and shifting the letters renders graphics glyphs. Up until now, we have focused completely on the regular character set. Now we'll take a look at how C64List handles programs that use the alternate (upper/lowercase) character set.

The alternate character set has the strange effect of rendering all normally uppercase letters as lower case, and their shifted counterparts as uppercase letters instead of graphics characters. In reality, the only thing this changes is what is visible on the screen. Although programming in this character set is done using all lowercase characters, the character codes are identical to the ones used in the normal character set. One thing that is very odd about this mode is that the character codes map very differently than the standard ASCII codes. This means that if a C64 program which is intended for mixed case (using the alternate character set) is converted to text using C64List, all lowercase characters will find themselves uppercased in the C64List text file, and all uppercase characters will become un-editable characters. And vice versa, when you type uppercase characters in the text format, they show up as lowercase characters in the .prg file, and lowercase characters convert to non-alphanumeric glyphs.

C64List is given no indication when converting .prg to .txt files, whether the characters are intended for the normal or alternate character sets. However, when developing software in the text format that is intended for mixed-case on the C64, C64List can do the conversion for you. To indicate that a program is meant for mixed-case on the C64, place the **{alpha:invert}** directive in-line in your code. This affects all alpha characters inside quotes in the program. It will cause the case of all alpha characters to be inverted, so they show up correctly when listed on the C64 in the alternate character set.

To turn off the invert function, use the directive **{alpha:normal}**. This will return the alpha character handling to normal. You can change the alpha handling mode wherever you wish, and however often you want, in your code.

There are also two other alpha settings available. One is **{alpha:upper}**, which will automatically convert all quoted lower-case characters to uppercase. This is great if you like to type in lowercase on your Windows machine, but want your program's output to be readable when using the standard character set. The other option is **{alpha:lower}**, which you may or may not find a use for.

When converting from .prg to .txt format, you may also use the `-alpha:normal|upper|lower|invert` command line parameter to convert the program to your desired .txt format. Note that only the normal and invert modes are very useful going this direction. You will probably want to leave the alpha mode set to Normal, unless you are working with a program that uses mixed case—in that case you will want to set the alpha mode to Invert.

Graphics glyph support

Another new feature starting with release 2.30, is the ability to handle graphics characters in a better fashion. Previously, C64List would simply port the C64 non-displayable graphics characters directly to ASCII and display whatever character resulted. This was not very handy for knowing what the characters were, and made it impossible to edit. Now, C64List will handle quoted characters in a very similar manner to how the unknown BASIC tokens are handled. For example, now when C64List encounters a shifted A character—a keyboard graphic character—it will convert the character to its associated numerical code and place it within curly braces to show that it is a token **{\$61}**. Note that this is for Normal alpha case. When you are working with the alternate character set you will want to specify alpha mode Invert and then it will come through as an uppercase A.

Similarly, when you convert from .txt to .prg, C64List will convert any numerical tokens of the form **{\$61}** to the actual character. You are free to use the numerical form for any of the characters you wish to insert. For example instead of typing a normal B, you could type in **{\$42}** instead, and C64List will convert it to a B.

C64List provides another input mode that can help you program graphics glyphs. If you know the keystrokes that provide the graphic character you are looking for you can type it in that way instead of the numeric token. For example, the spade character is a shifted A on the C64. If you know this, you can insert a spade glyph into your code this way: **{shft}A**. The **{shft}** directive tells C64List that you have virtually pressed the shift key and any letters you type will be shifted. The **{cmdr}** and **{ctrl}** directives similarly tell C64List that you have virtually pressed the Commodore key or control key. The current shft/ctrl/cmdr directive is in effect until another such directive is encountered, the end of the line is

reached, or a {lift} directive is encountered.

Custom screen codes

Starting with release 2.30, you may add, modify and remove custom screen (i.e., cursor) codes. The {quoter} directive gives you the ability to change the screen code tokenizer in the same way that {tokenizer} allows you to change the BASIC tokens. For example, if you would like to replace the shifted A ♠ glyph with the word {spade}, you can do it like this:

```
{quoter:$61="spade" }
```

After executing the above directive, whenever C64List encounters {spade} within quotes, it will happily convert it to character \$61, the spade character. As with the BASIC tokens, you may add these directives into a tokenizer file so conversions from .prg to .txt will also enjoy this ability.

Release 2.30 also introduced some new screen codes for the symbols that are present in the C64 character set that do not correspond to ASCII characters: {up arrow}, {back arrow}, {pound}, {shft pound}, {ctrl pound}, and {pi} are all new.

Including files

If you would like to split your source code into multiple files, C64List has two ways of supporting this.

The {include:<filename>} directive inserts the specified file at the location in the code where the directive exists. 50 levels of inclusion are allowed. You must avoid circular includes, otherwise C64List will complain. For example, if a.txt includes b.txt, and b.txt includes a.txt, this will cause an error.

Alternately, the {uses:<filename>} directive is very similar to {include}, but it is smarter about things. When C64List encounters a {uses} directive, it first checks to see if the specified file has already been included. If not, it parses the specified file; if however, it sees that the file has already been parsed, it will silently ignore the request to use the file. This is superior to {include} for a few reasons:

- 1) It automatically prevents circular inclusion
- 2) It allows multiple files to include the same other file without complaining about it
- 3) Since multiple files can use the same third file, it eliminates a lot of file order dependencies

C64List keeps track of which files include other files, and where they are located in the directory structure. If a file {include} or {uses} another file and do not provide the path to the file, C64List will look for the file in the same directory of the file that is including/using it.

Conditional compiling

C64List provides a means of conditional compiling through the use of the following directives:

```
{def:<variable name>}
{undef:<variable name>}
{ifdef:<variable name>}
{ifndef:<variable name>}
{else}
{endif}
```

These directives allow blocks of code to be ignored by C64List under various conditions. You can **define** or **undefine** a variable, and then use an {ifdef} ... {endif} pair (or other various forms) to delineate the conditional code.

Only one conditional block can be in effect at any given time in the code. In other words, nesting of {ifdef}s is not allowed.

Note that these directives are in the BASIC domain, so they do not work inside {asm} blocks. If needed, you can exit an {asm} block with {endasm}, add a conditional compile directive, then reenter assembly mode. This restriction may be removed at some point.

Currently, only one conditional compiler directive may be present on a given line of code, and nothing else may be on the line with it.

C64List Disassembler

C64List allows machine-language .prg files to be disassembled. By default, C64List looks at the load address of the .prg file and begins disassembly at that address, and continues to the end of the file.

Note that as with any disassembler, it will simply attempt decode bytes as if they were valid assembly instructions. Therefore, although you may attempt to disassemble a BASIC program, the result will be junk. Also, inline data can cause the disassembly to get off track, since a disassembler has no way of identifying data versus code.

C64List does have tools to help you get a meaningful disassembly listing however. For one thing, you may specify a starting and ending address for the disassembly using the `-range` command line parameter. Some examples:

- 1) List assembly instructions starting at \$8020 and continue to the end of the file
`C64List asm.prg -range:$8020`
- 2) List assembly instructions between 8192 and 8265
`C64List asm.prg -range:8192-8265`

Note that the `-range` command line parameter also works for hex dumps.

Another tool that can help to digest unfamiliar code and make it more readable, is C64List's label-stuffing feature. Using the `{-tokenizer}` directive you may specify a file that contains symbol assignments. So as you identify subroutines and other interesting addresses, you can name them by adding symbol assignments to the tokenizer file. For example if you find a function at \$3561 that writes a string to the screen, you might want to name it "printstring". To do this, add the following to the tokenizer file:

```
printstring = $3561
```

Now, next time you disassemble the file, the label "printstring" will appear at location \$3561. In the near future I also plan to insert the symbol names into the operands as well.

Another feature is the `-con` command line option. This will direct C64List to output hex and asm output to the console instead of a file. It has no effect on tokenized or text BASIC output.

C64List Assembler

As of version 3.0, C64List contains an assembler. The assembler feature is in a rather early stage, so there may currently be bugs and not all planned features or syntax exist yet.

To enter the assembler, use the `{asm}` directive. Likewise, `{endasm}` exits the assembler. C64List's assembler is built so that BASIC programs can easily call assembly routines that are embedded in the same file, after the end of the BASIC code. All the BASIC code should come before any assembly code. (Note: C64List does not currently enforce this rule.) C64List automatically finds the end of BASIC and locates the assembly code immediately after the final NULL link.

Assembly functions should be labeled with a symbol so that they can be called from BASIC or other assembly routines. To access an assembly symbol function from BASIC, use the `{sym:<symbol name>}` directive. The `{sym}` directive will be replaced in the code by the associated address. For example, to call an assembly function, use `sys {sym:FunctionName}`; to read a byte from memory, use `peek({sym:ByteName})`. Here is a very brief example calling an assembly function from BASIC in C64List:

```
sys {sym:AssemblyFunction}
{asm}
Screen = 1024
Checker = $ff
AssemblyFunction:
    lda Checker
    ldx #$28
L0:
    sta Screen,x
    dex
    bpl L0
    rts
```

General Info

- Only one statement is allowed per line
- All assembly-language instructions must be placed inside `{asm}` and `{endasm}` directives
- There can be any number of `{asm}` and `{endasm}` directive pairs
- No BASIC code is allowed after the first `{asm}`

- {asm} and {endasm} must appear on lines by themselves
- All directives are in the BASIC domain, which means they must not be inside an {asm} block.
- No special command line options are required to assemble code.

Symbols and Labels

- Assembly labels are added as symbols into the symbol table
- The assembly symbols are completely separate from the BASIC line labels
- To define a symbol, use the format `<symbol> = <value>` like this:

```
Screen = 1024
```
- Values may be written as decimal or hex
- Symbols are case insensitive
- Symbols may only be defined once; any attempt at redefinition will produce an error
- Values may be expressed either in decimal or hexadecimal
- To define a label, add a new name that has not yet been defined and follow it with a colon

```
Label1: lda #$00
```
- A label may be placed on a line with code (as shown above), or on a line by itself:

```
Label2:
    lda #$01
```
- Multiple labels may be created for the same line of code:

```
Label3:
Label4: lda #$ff
```
- Use the `-sym` command line option to dump the symbol table after assembling.

Operands and Expressions

Symbols, instructions, and pseudo-ops may require an operand. Operands are somewhat dependent on what they are attached to. However, in general, an operand is numeric, and often may be an expression. C64List provides a number of elements that may be used in expressions:

Symbol	A symbol from the symbol table may be used; some restrictions apply to labels.
Number	A numeric value may be specified
+	Adds two elements of an expression
-	Subtracts two elements of an expression
&	ANDs two elements of an expression
	ORs two elements of an expression
<	Low byte of expression (may only appear at the beginning)
>	High byte of expression (may only appear at the beginning)

Zero Page addresses

The 6510 processor has a number of instructions that utilize various “zero page” addressing modes. This means that addresses in the range 0-255 can be either addressed in absolute modes (requiring two bytes), or zero page mode (requiring one byte). The way to tell C64List that you want to use a zero page address is in the way you write the value. One- or two-digit hex values symbolize a zero-page address, while three- or four-digit hex values symbolize absolute addresses. For decimal values, 1,2, or 3 digit values that are 255 or less identify a zero page address, while 3-digit values larger than 255, or greater than 3 digit values identify absolute addresses.

Examples of zero page addresses:

```
$4
$0A
$ff
255
```

Examples of non-zero page addresses:

```
$100
256
$000A
0255
```

When a symbol is defined, the zero page attribute follows it. For example,

```
Zp = $0F      ;Zp is a zero-page symbol
nZp = $00F    ;nZp is an absolute symbol
Zp2 = Zp+1    ;zp2 is a zero-page symbol
```

When assembly instructions are assembled into machine code, C64List evaluates the zero-page attribute of the operand and assembles the correct opcode for the addressing mode implied by the operand.

Therefore, using the above symbol definitions C64List would do the following:

```
lda Zp      ;do a zero-page lda from address $0f
lda nZp     ;do an absolute lda from address $000f
```

This also implies that an error will occur if instructions which can not accept zero page addresses are presented with an operand specified as zero-page. For example the following would cause an error:

```
jsr $45
```

Labels used in symbol assignment

There is currently a somewhat annoying limitation placed on labels, in that they can't be included as any part of a label assignment. For example, this would fail:

```
Label0: rts
LastInstruction = Label0
```

The reason is that although labels are defined as soon as they are encountered in the code, their actual value can't be assigned until very close to the end of the assembly process. Symbol assignments, however, are done as soon as they are encountered. This means that by the time labels are assigned a value, all the symbols have already been defined. I will attempt to remove this restriction in the future.

Pseudo Ops

The following pseudo-ops are currently defined:

byte <*n*> ;insert a single byte of value *n* in the code stream
word <*n*> ;insert the two-byte value *n* in the code stream
area <*n*> ;insert *n* bytes of space into the code stream, with undefined values

The above pseudo-ops may also be used in combination with labels, so the addresses of these variables can be accessed from assembly and BASIC.

Including and Using assembly files

Since {include} and {uses} are directives, they may only appear *outside* of {asm} blocks. This restriction may eventually be removed, but at this time, the recommended method of including files is to make each file have its own {asm}{endasm} block, as follows:

File a.txt

```
` comments
{asm}

Return: rts

{endasm}
```















File b.txt





























```
`comments
{uses:a.txt}

{asm}
    jmp Return
{endasm}
```

Reference Tables

The cursor control codes recognized by C64List are listed in the following table. All of the listed cursor control codes may be suffixed with a colon followed by a repetition value (such as "{down:10}")

Control code	Glyph	Control code	Glyph
Change printing color			
CTRL + 1 {black}		C= + 1 {orange}	
CTRL + 2 {white}		C= + 2 {brown}	
CTRL + 3 {red}		C= + 3 {lt. red}	
CTRL + 4 {cyan}		C= + 4 {gray1}	
CTRL + 5 {purple}		C= + 5 {gray2}	
CTRL + 6 {green}		C= + 6 {lt. green}	
CTRL + 7 {blue}		C= + 7 {lt. blue}	

Control code	Glyph	Control code	Glyph
CTRL + 8 {yellow}		C= + 8 {gray3}	
CTRL + 9 {rvrs on}		Invert the printing colors	
CTRL + 0 {rvrs off}		Restore the normal printing colors	
Cursor control			
{down}		Move cursor down	
{up}		Move cursor up	
{left}		Move cursor left	
{right}		Move cursor right	
{home}		Move cursor home	
{clear}		Clear the screen and move the cursor home	
{insert}		Insert mode character (printable insert key)	
{delete}		Insert mode character (printable delete key)	
{shft ret}		Shifted return	
{stop}		Insert mode character (printable stop key)	
Function keys			
{f1}		Function key F1	
{f2}		Function key F2	
{f3}		Function key F3	
{f4}		Function key F4	
{f5}		Function key F5	
{f6}		Function key F6	
{f7}		Function key F7	
{f8}		Function key F8	
Keys not available on a PC			
{up arrow}		The up arrow character	
{back arrow}		The back arrow character	
{pound}		The British Pound symbol	
{shft pound}		The character you get when shifting the pound key	
{ctrl pound}		The character you get when hitting the pound key + ctrl	
{pi}		The pi symbol	

The directives recognized by C64List are listed in the following table.

Directive	Description
{renumber}	Request that the following BASIC program be renumbered. Restriction: must be placed before any BASIC code in the file.
{number:<line number>}	Causes the next line of code to have the specified line number.

Directive	Description
	<p>Restrictions:</p> <ul style="list-style-type: none"> The requested line number must be greater than any line number encountered so far in the file; if this requirement is not met, the directive will be ignored. The specified value must be a valid line number for the C64
{step:<value>}	<p>Causes the line numbering to increment by the specified value. Activates on the second line of code after the directive.</p> <p>Restrictions:</p> <ul style="list-style-type: none"> Must be positive Must be small enough to allow all lines in the program to be assigned valid line numbers
{nice:<value>}	<p>Causes the next line number to be “niced” to a multiple of the given value. Typically, the specified value should be some factor of ten. For example, if the next line number to be assigned was 437, specifying {nice:100} will instead cause the next line number to be 500.</p> <p>Restriction: Must cause the next line number to be within the valid range.</p>
{crunch:<switch>}	Turns on or off crunch mode.
{remremoval:<switch>}	Turns on or off the REMark removal in-line in your BASIC code.
{loadaddr:<address>}	<p>Request that the following BASIC program be renumbered.</p> <p>Restriction: must be placed before any BASIC code in the file.</p>
{tokenizer:<tkn>=<kywrd>	Adds custom BASIC tokens for BASIC extensions.
{quoter:<tkn>=<kywrd>	Adds custom screen tokens for characters in quote mode
{alpha:<mode>}	<p>Sets the alpha character handling mode, which tells C64List how to handle printable characters that are inside quotes. The settings are:</p> <ul style="list-style-type: none"> None (default: characters are not converted) Upper (convert lowercase characters to uppercase) Lower (convert upper characters to lowercase) Invert (invert the case of all alpha characters)
{include:<file>}	When loading text-formatted files, includes the contents of another text-formatted file.
{uses:<file>}	Similar to {include}, but automatically checks to see if the file has already been included. If so, does not include it again.
{:<label>}	A label starts with a colon
{sym:<symbol>}	Refer to a symbol in the assembly symbol table
{asm}	Start assembler mode
{endasm}	End assembler mode
{def:<variable name>}	Defines a conditional compiler variable
{undef:<variable name>}	Undefines a conditional compiler variable
{ifdef:<variable name>}	Begins a block of lines which are only processed if <variable name> is currently defined (using {def})
{ifndef:<variable name>}	Begins a block of lines which are only processed if <variable name> is currently not defined .
{else}	Ends an {ifdef} or {ifndef} block and immediately starts another

Directive	Description
	block with the opposite condition
{endif}	Ends an {ifdef}, {ifndef}, or {else} block and returns to normal processing.

C64List syntax

C64list <input filename> [options]

The following command line options are available in C64List:

Command Line Option	Description
-prg[:filename[.ext]]	Convert the loaded program to .prg format. The default name will be <input filename>.prg if not specified here.
-p00[:filename[.ext]]	Convert the loaded program to .p00 format. The default name will be <input filename>.p00 if not specified here.
-hex[:filename[.ext]]	Output the loaded program as a text formatted hex dump. The default name will be <input filename>.hex if not specified here.
-asm[:filename[.ext]]	Disassemble and output the loaded machine language program in text format. The default name will be <input filename>.asm if not specified here.
-txt[:filename[.ext]]	Convert the loaded tokenized BASIC program to readable ASCII text format. The default name will be <input filename>.txt if not specified here.
-lbl[:filename[.ext]]	Convert the loaded tokenized BASIC program to readable ASCII text format, remove all line numbers, insert labels where necessary, and change all GOTO/GOSUB references to these labels. The default name will be <input filename>.lbl if not specified here.
-crunch	When converting to a tokenized binary BASIC format, remove all unnecessary spaces. When converting to an untokenized ASCII format, strategically add spaces to make the code more readable
-crsr	When converting to text, retain the binary cursor codes instead of converting them to editable strings. When converting to tokenized BASIC, has no effect; the converter will automatically convert either format correctly.
-rem	When converting to tokenized BASIC, remove all REM statements. When converting to text, has no effect
-ovr	Overwrite an existing file if necessary
-f	Same as -ovr
-verbose[:<mode>]	C64List will output much more information about what it is doing while it is loading, converting and saving files. This is useful if you are doing something incorrectly and need hints to find out what is wrong. Specifying -verbose only is the same as -verbose: on. Not specifying -verbose is the same as -verbose: off. You may also specify -verbose: list, which outputs each line of text when converting to

Command Line Option	Description
	.prg/.p00.
-alpha:<mode>	Sets the alpha mode on the command line. May be set to one of: Normal Upper Lower Invert.
-range:<start>[-<end>]	When outputting .hex or .asm files, this allows you to limit the memory range which is output to the file.
-h	Output a help screen and exit immediately
-tokenizer:<filename>	Specifies a text-formatted file containing tokenizer directives. This function is necessary for loading programs containing customized tokens from a tokenized format. When loading from text, the custom tokens may also be loaded this way, but it is recommended to put the tokenizer directives in the source file itself.
-b4	Enable the BASIC 4.0 extensions
-b7	Enable the BASIC 7.0 extensions (single byte token support only)
-con	Debug tool: sends the output for hex and asm (only) formats to the console instead of any specified .hex or .asm file.
-sym	Dumps the symbol table to the console after all code has been tokenized and assembled.

Future plans

C64List will probably continue to evolve, as time, interest, and other factors permit. Some of the tentative features planned are:

- .d64 file support
- Loading .hex formatted files

C64List versions

This document was released with C64List version 2.00

V1.00—First working version was able to load .prg, .p00 files and convert them to .txt and .hex files. It also had cursor code recognition and an unsafe version of the REMark removal feature.

V2.00—First official release; added a lot of features, including

- Line renumbering
- Line denumbering (convert to labeled format)
- Load labeled text format
- GOTO/GOSUB re-targeting
- Safe REMark removal
- BASIC code relocation (alternate load addresses)

V2.01—Added some smaller features

- Text-only line comments: start with ' character and are removed as they are loaded
- Added {remremoval:on|off} in-line REMark removal switch
- Added {tokenizer:token1="keyword1",token2="keyword2"...} directive

V2.20—New directives and command line options

- Verbose command line switch
- Turned off verbose mode by default
- {crunch:xxx} directive
- {alpha} directive
- Automatically converts unquoted lowercase characters to uppercase (i.e. variables)

V2.21—Bugfix for REM removal with line numbers problem

V2.22—Bugfixes & enhancements

- (re)enabled illegal instruction display in asm mode
- Added a -tokenizer:<file> command line parameter
- Added tokenizer support for prg->txt conversions via tokenizer file
- Bugfix: ticks inside DATA and REM statements no longer count as tick comments.
- Labels are now case-independent
- Fixed line 0 is out-of-order bug
- Hex dump characters now show reversed and lower case characters
- Added -b4 & -b7 parameters to help BASIC 4.0 & 7.0 developers
- Fixed an asm branch off-by-two bug (PC pre-increment)

2.23Beta23-Aug-2011—(Released to Agent Friday only)

- Produce error when NUMBER directive can't be honored (but assign a higher number anyway)
- Produce verbose output when renumbering is automatically initiated
- Added help for newer cmd line params
- Require load filename to be first parameter on cmd line
- Earlier parsing for verbose parameter
- Fix for "marking" cmd line params (doesn't directly affect C64List)

2.23—30-Aug-2011 (Bug fixes)

- All fixes from 2.23 Beta above
- Fix for linking problem due to zero-length lines

2.30—22-Sep-2011 (Cool new stuff)

- new graphics character support:
 - created a "Petskey" to "Petscii" converter (Petskey is a C64List invented code giving a readable ASCII character and a shift/control/cmdr code that can translate directly to a C64 graphics character.
 - reads {shft}abc{lift} syntax
 - reads {\$xx} syntax for any character
 - writes {\$xx} syntax for unrecognized/undisplayable tokens
- new BASIC token support:
 - reads {\$xx} syntax for any token
 - writes {\$xx} syntax for unrecognized tokens
- added some tokens for non-ASCII characters to the screen tokens list:
 - "up arrow"
 - "back arrow"
 - "pound"

- "shft pound"
- "ctrl pount"
- "pi"
- added {quoter} directive for allowing custom screen tokens, much like {tokenizer} does for BASIC tokens
- added ability to remove existing tokens from both the {tokenizer} and {quoter} lists. Specify a blank value to remove a token
 - ex: {tokenizer:\$80=""}
 - or {quoter:\$91=""}
- bugfix: an error message about unknown directives was lost during a previous check-in. This has been restored.
- displays the original line contents when an error occurs in text-to-prg tokenizing, when possible
- Added a new verbosity level and command line parameter support
 - (no verbose parameter)--same as before
 - verbose--same as before
 - verbose:off--same as no verbose parameter
 - verbose:on--same as -verbose
 - verbose:list--outputs incoming line of text when converting from text or label format to tokenized
- Added a line of information at the end of C64List telling how many errors and how many warnings were encountered.
- added an optional -range:xxxx-yyyy command line parameter that is used for outputting .asm and .hex files:
 - range:xxxx--starts at address xxxx and goes to end of file
 - range:xxxx-yyyy--starts at address xxxx and ends at yyyy
 - bugfix: dont crunch/uncrunch data statements
 - bugfix: don't trigger inData mode if the token for DATA is encountered in a quoted string
 - now supports Alpha mode settings on the command line for conversions from .prg -> .txt
 - bugfix: the pi BASIC keyword and screen token were not enabled

3.00—08-Jan-2012 Assembler, lots more

- built-in 6510 assembler that integrates with BASIC. Assembled machine code is placed in memory after the BASIC code.
- All 256 opcodes (both normal and undocumented) are supported
- {asm} and {endasm} directives start/end assembly mode.
- Symbol table: labels and other symbols are placed in the symbol table
- {sym:<symbol>} directive accesses a symbol table symbol from within BASIC. ex.


```
sys{sym:subroutine}
```
- Symbol table export: add -sym to your command line to get a symbol table dump
- The symbol table export contains = signs between the symbol and its value, so you can cut and paste the contents and use it directly in code
- "byte" and "word" pseudo-ops are supported
- Operands may be numeric (decimal or hex "\$"), or may reference assigned symbols.
- Operand resolution handles the following:
 - immediate values "#"
 - symbols (use the symbol name)

- addition (example: screen +40)
- subtraction (example: endofscreen -40)
- and (example: flag1 & flag2)
- or (example: flag1 | flag2)
- low-byte only (example: lda #<screen)
- high-byte only (example: lda #>screen)

-Symbols are now allowed in the -tokenizer file; this allows disassembled code to contain labels where appropriate.

-New directive {assign:label=linenumber} that allows you to reference labels that do not exist in the source code.

-Added conditional compile directives: {def} {undef} {ifdef} {ifndef} {else} {endif}

-Added {uses:file} directive; use this feature instead of {include} to avoid the standard {includes} problems of recursion and multiply included files

-Added checking to prevent labels from being defined more than once

-{lift} wasn't working completely; fixed it

-{cmdr} keys were mistakenly assigned to {ctrl} keys

Notes/Limitations:

-All directives are handled in BASIC mode, so you must switch out of assembly mode in order to use them. For example, if, in

one assembly file you wish to include another, you must {endasm} before you {include}. I recommend that all source files start in BASIC mode, and have the

{asm} and {endasm} directives at the beginning and ending every file.

-each assembly instruction must appear on its own line

-The {asm} and {endasm} directives must be on their own lines, with nothing else

-The {def} {undef} {ifdef} {ifndef} {else} {endif} directives must be on their own lines, with nothing else.

-Only one {ifdef} {ifndef} {else} {endif} may be in effect at any given time (no nesting is allowed)

-word and byte pseudo-ops may only contain one value each. To define multiple bytes or words, you must specify additional byte or word pseudo-ops, each on their own line.

-labels can currently not be used in an expression when defining symbols (for example, if label: is defined, endOfPrg = label+1 will fail).

-the low-byte/high-byte notation in the operands must appear as the first element in the operand. For example:

lda #<screen +40 is legal

lda #40 + <screen is not legal. this is not checked for, but it will result in the

equivalent of "lda #<screen" (the 40 is forgotten)

-{assign} simply creates a label that is statically assigned to a given line number. It does not reserve the line number in any way, and does not affect renumbering. It is up to the user to ensure that the assigned line numbers do not get re-assigned to code within the source.