C64List 4.xx User's Guide
By Jeff Hoag

Contents

C64List 4.00 User's Guide

What is C64List?

C64List is a Windows command-line based tool to aid in cross-platform development of Commodore 64 software.

C64List converts and detokenizes Commodore BASIC .prg files into readable text files so they can be edited using your text editor of choice on a Windows machine. Once your text-formatted source code has been modified, C64List also re-tokenizes it back into a .prg file that can be loaded directly into a C64 machine or simulator such as Vice. C64List is also a symbolic assembler for writing and assembling your own 6510 machine language programs, as well as a 6510 code disassembler, and file dump-to-ASCII hex utility.

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

History

2008: C64List version 1.00 released (original release)
2010: C64List version 2.00 released
2011: C64List version 3.00 released
2019: C64List version 4.00 released.

Starting around 2012 it became obvious that the assembler was lacking in some needed features. The assembler feature was on top of, and intertwined with the BASIC tokenization/detokenization code which made it extremely difficult to maintain. It also prevented C64List from reporting meaningful errors, particularly while assembling assembly language code. So rather than add to the existing difficult-to-maintain codebase, a new stand-alone assembler project called "casm" was developed as an independent project and started from the ground up. Once casm became mature, it was a lengthy project to cleanly re-integrate the BASIC handling code. C64List version 4.00 is the culmination of this work. Version 3.xx users will find version 4.00 much more code-developer friendly.

New in 4.00 highlights

Version 4.00 is a completely new code base from previous revisions. The code was refactored into three major, mostly independent modules:

1. Source loader/parser/preprocessor. Code being read from a file first passes through this module, which handles loading both BASIC and assembly code, tracking source file line numbers and error reporting.
2. Casm – 6510 assembler
3. BASIC tokenizer/detokenizer.

This is an important concept to understand because some features and directives are specific to certain

modules and not applicable (or behave differently) in others.

Some C64List 4.00 improvements of note:

- Errors are now reported with a file name and line number in source code for easy identification
- Improved and expanded alpha conversion functions
- User configurable upper/lowercase options for .prg-to-BASIC conversion output.
- Line numbers for labeled lines are preserved by default when converting to .lbl format
- New directives
- Hexadecimal and binary values can be inserted directly into code using directives
- Pre-processor source-loader phase
- More than one directive is allowed on a single line of code
- New command-line options
- Improved output-to-console options for text formatted file types
- Much improved assembler features, including
    - Assembler directives
    - New pseudo ops
    - Math and labels are allowed in all instruction and pseudo op operands
    - More flexible parameter options in pseudo ops such as **byte** and **ascii**
    - Local, general, and important symbols
    - Nameless local labels
    - Supports alpha conversion and C64List-style strings in assembly code
    - Numeric expressions are evaluated algebraically rather than left-to-right
    - Parentheses are accepted and evaluated algebraically in expressions
    - The asterisk ("*") as an expression term identifies the current address being assembled
    - The question mark expression terms ?,??,???,???? introduce random 4,8,12,16-bit values into expressions
    - Most error messages include reference to source file and line number

This documentation is specific to C64List 4.00 and incremental revisions. While versions 4.xx are mostly compatible with the 3.xx line, some of the behaviors are slightly different. The main thing to watch out for when moving your code from 3.xx to 4.xx is in the assembler: please ensure that all expressions follow algebraic precedence rather than assuming left-to-right.

How to use C64List

For code that is to be ported from existing Commodore 64 projects, the first step to cross development is to devise a way to move files from a 1541 disk into the Windows file system. There are numerous ways of doing this including the using Vice64 emulator, the X1541 cable solution, or over the internet using CommodoreServer.com using a Comet64 modem or similar device. Once your file is in .prg format on the local machine, you can use C64List to convert it to a plain text-based file. C64List also supports loading from and saving programs into .d64 files.

The development model that C64List advocates is *to-text once, to-.prg many times*. In other words, make the to-text conversion only once--at the beginning of development. Once this is done, all further development is made using the .bas code file, and conversions for the source file to an executable .prg file can be made as many times as necessary. Of course, if you are starting a new project from scratch, there will be no .prg file to convert--you can simply create a new plain-text .bas source code file.

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.

Getting help

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

```
C64List
```

With no parameters and C64List will print out a help screen. Specifying the help parameters `-h` or `-help` or `-?` will do the same thing. The nice thing about using an explicit option is that the help parameter takes precedence over all other parameters: if you are typing a long command line but need help remembering some option name, you don't have to erase everything you've already typed—just add `-h` and C64List will give you the help and nothing else. Here's an example of the help screen:

```
--------------------------------------------------------------------------------
                            C64List 4.00 Beta A
                      Copyright (c) Jeff Hoag 2008-2019
--------------------------------------------------------------------------------
 Loads Commodore 64 BASIC programs and converts them between various formats.
 Automatically determines the format to load by the file extension.
 Loads these file types and switches to the appropriate mode:
   txt|lbl|bas|d64|prg  Enters BASIC tokenizer/detokenizer mode
   asm                  Enters CASM assembler mode
--------------------------------------------------------------------------------
Syntax:
        c64list <filename.ext> [-<options>]
 Control parameters include:
        -ovr                    overwrite existing files
        -verbose[:list]         output additional information during conversion
```

```
        -def:<var[=val]>        define one or more parser variables

---- Tokenizer mode ----
        -crunch                 remove unneeded spaces
        -rem                    remove all REM statements
        -labels                 dump a list of defined labels to screen

---- Detokenizer mode ----
        -autospace              add spacing for readability
        -crsr                   preserve binary cursor codes
        -keycase                output keywords in lowercase
        -varcase                output variables in lowercase
        -pref:name              load a preface file
        -alpha:<mode>           ascii|lazy|alt|upper|lower|poke
        -supervariables         create and use Supervariables

---- Assembler mode ----
        -symlvl !|-|@|auto      filter .sym output important|general|local
        -sym3                   output symbol table for c64List3.x consumption

---- Output formats ----
        -txt[:name]             save untokenized BASIC in a text format
        -lbl[:name]             save untokenized BASIC in a labelized text format
        -prg[:name]             save in .prg format
        -hex[:name]             save prg in an ASCII hex dump format
        -bin[:name]             save binary file with no load address
        -lst[:name]             save in a memory-disassembled format
        -sym[:name]             output a symbol table file

        (load from .d64):       "d64file[.d64]::C64 FILE NAME"
        (save to d64)           -d64:d64file[.d64]::C64 FILE NAME"

---- Miscellaneous ----
        -range:<start>[:<end>]  output address range for .hex/.lbl files
        -notbasic               loaded .prg file does not contain BASIC

Contact: c64List@gmail.com
```

C64List syntax

C64List is invoked with one mandatory parameter, the input filename, followed by optional command line parameters. See the section entitled *C64List command line parameters* for a list of all possible parameters. There must be at least one output format specified, or C64List will complain.

```
C64list <input filename> [options]
```

The exception to this rule is to output the help screen. Entering C64List with no parameters will display the help screen and exit. Alternately, entering C64List with one of the help parameters (-h, -help, or -?) will also display the help screen and exit, while ignoring all other parameters.

Part 1: BASIC

The first part of this document focuses on C64List's BASIC language features.

Quick start: Converting a BASIC program to plain-text source format

We will go over C64List's full syntax later, but just to get things moving quickly, let's start out with an example. Let's assume that you have an old BASIC program named `Example1` that you wrote years ago and now you want to add something to it. You successfully extracted it from the old floppy disk or tape, and now it's in a file on your PC called `Example1.prg`. If you try to load this file into an editor as-is, you would find that it is very garbled. That's because the file contains tokenized C64 BASIC code and not just plain text. Since the file is no longer on a real C64, you can't simply LIST it. Windows needs it to be converted to a plain-text format first. This is where C64List comes in. To do this, you would type the following command:

```
C64List Example1.prg -prg:NewFile -hex –txt –lbl:Example1.bas
```

Normally, you'd just save the output to a single file, but this example converts it to four different formats to show the various options that C64List provides.

Of course, C64List.exe is invoked in the command line first. After that, the first parameter is specified without any leading "`-`". This is the name of the file that you wish to convert. Here, we load "`Example1.prg`". C64List uses the extension of the input file to determine what format to expect, so in this example, it knows we are loading a tokenized, BASIC program file.

The remaining parameters tell C64List what formats to output. The name of the input file is propagated to the target filename and appended with the specified extension. For example, `–txt` tells C64List to convert the loaded file into text format and save it as "`Example1.txt`".

If you accidentally forget to give the .prg file a different name, C64List will notify you that you are trying to overwrite your input file ("`Error: Output file would overwrite input file`"), and then stop so your input file does not get overwritten.

In a single command line, C64List will output up to one of each file format that it supports. In our example, we requested C64List to output four different files in four different formats:

- `NewFile.prg` (another tokenized BASIC file that should be identical to the original)
- `Example1.hex` (a hex-dump format), and
- `Example1.txt` (a text file)
- `Example1.bas` (a specially-formatted text file, ideal for development)

Now that we've generated these files, let's examine each of them in more detail.

`NewFile.prg`

Notice that we told C64List to name the output .prg file with a different name to avoid conflicting with our original input file name. The new filename is specified by typing the new file after the output format specifier, separated by a colon.

Why would we want to read a .prg file and output the same file contents to another file? Typically you won't need to do this, but it's an interesting exercise. Behind the scenes, C64List actually internally converts the file to text, then re-tokenizes it back into the C64 BASIC executable format. Since we didn't tell C64List to do anything exotic, `NewFile.prg` will be identical to `Example1.prg`. Some advanced techniques, such as line number renumbering can be done in a single command line, in which the output .prg file will be different from the input file. The content of this file is mostly unreadable by humans since it is tokenized so we won't actually look at it here. The .hex file (below) is much more readable so we'll look at that next.

Example1.hex

```
0801: 33 08 64 00 8f 20 54 48 49 53 20 49 53 20 41 20|3.d.. THIS IS A
0811: 42 41 53 49 43 20 50 52 4f 47 52 41 4d 20 46 4f|BASIC PROGRAM FO
0821: 52 20 54 45 53 54 49 4e 47 20 43 36 34 4c 49 53|R TESTING C64LIS
0831: 54 00 51 08 6e 00 54 57 20 b2 20 31 30 20 3a 20|T.Q.n.TW . 10 :
0841: 48 49 24 20 b2 20 22 90 93 05 11 11 11 11 22 00|HI$ . ".......".
0851: 7e 08 78 00 99 20 48 49 24 20 22 54 48 49 53 20|~.x.. HI$ "THIS
0861: 49 53 20 41 4e 20 45 58 41 4d 50 4c 45 20 42 41|IS AN EXAMPLE BA
0871: 53 49 43 20 50 52 4f 47 52 41 4d 22 00 92 08 82|SIC PROGRAM"....
0881: 00 99 20 22 46 4f 52 20 54 45 53 54 49 4e 47 22|.. "FOR TESTING"
0891: 00 ad 08 8c 00 99 20 22 1d 1d 1d 1d 1d 1d 1d 1d|...... ".......
08a1: 1d 1d 43 36 34 4c 49 53 54 21 22 00 bc 08 96 00|..C64LIST!".....
08b1: 54 57 20 b2 20 54 57 20 ab 31 00 d1 08 a0 00 8b|TW . TW .1......
08c1: 20 54 57 20 b1 20 30 20 a7 20 89 20 31 34 30 00| TW . 0 . . 140.
08d1: d7 08 aa 00 80 00 00 00                         |........
```

This lists the output .prg file in hexadecimal codes. This is what the file looks like when it is inside the C64's memory. You can read some of the strings, but there's a lot of nonsense mixed in. That's the tokenized BASIC code we talked about earlier. Beginner-level BASIC programmers can ignore this format if they wish, but this is a very useful output format for some advanced programming and debugging.

Example1.txt

This should look pretty familiar. It's what you would see if you LISTed the program on a C64.

```
100 REM THIS IS A BASIC PROGRAM FOR TESTING C64LIST
110 TW = 10 : HI$ = "{black}{clear}{white}{down:4}"
120 PRINT HI$ "THIS IS AN EXAMPLE BASIC PROGRAM"
130 PRINT "FOR TESTING"
140 PRINT "{right:10}C64LIST!"
150 TW = TW -1
160 IF TW > 0 THEN GOTO 140
170 END
```

Sharp-eyed readers will notice that the cursor control codes look very different from what you'd see on a C64 listing. Instead of the familiar, but cryptic reverse-character control codes, here we have nice, readable codes like {black} and {clear}. C64List does this for a few reasons. First, it's far easier to read than the C64-style codes. More importantly, most Windows editors are not able to display those special characters, and even if they could be displayed, Windows keyboards don't have those keystrokes. Finally, C64List also compresses repetitive control characters, so instead of {down}{down}{down}{down}, it outputs {down:4} which is much more readable and manageable.

If you wish, you can use the code in this file to begin your development. C64List is able to convert this format right back into a .prg file that can be run on a C64. However, C64List has more nifty tricks up its sleeve to make programming even easier. Check out the next output file for details.

Example1.bas

```
      REM THIS IS A BASIC PROGRAM FOR TESTING C64LIST
      TW = 10 : HI$ = "{black}{clear}{white}{down:4}"
      PRINT HI$ "THIS IS AN EXAMPLE BASIC PROGRAM"
      PRINT "FOR TESTING"
{:140}
      PRINT "{right:10}C64LIST!"
      TW = TW -1
      IF TW > 0 THEN GOTO {:140}
      END
```

This is the file output due to the –lbl:Example1.bas parameter. We renamed this file from the default .lbl extension to the .bas extension because we plan to use the source code in this file as the basis for our new project.

The format is almost the same as the previously described .txt format, except what happened to all the line numbers? If you have programmed BASIC code on a C64, you probably have discovered that line numbers present difficulties, especially when you need to move parts of your code around, or add code between line numbers. C64List lets you completely disregard all line numbers while you are developing you program, and it will automatically add them into your code when you convert your program back to a .prg for you. C64List advocates for all BASIC development to be done using this line-numberless format. There is much more information about this feature in later sections.

Quick start: Converting plain-text code to a runnable BASIC program

Once you have completed modifying your text-based BASIC source code, it is time to convert it to the .prg format so it can run on a C64. As it turns out, the C64List command line to do this is very similar to the previous exercise. Simply tell C64List to read your text formatted source file and save it as a .prg file, like this:

```
C64List Example1.bas –prg
```

If you followed the previous example, you will run into an error here. Your original file, `Example1.prg` is still in the directory, so C64List warned that you told it to overwrite your original program (`Error: File already exists: Example1.prg`). The best practice is to **create a work area in another directory, away from your original file** and move your new source file into the fresh work area and do all new development there. This will ensure your original file is safe from being overwritten. Once you have created your new work area and changed directories there, retry the command again. This time it should successfully save a new file called `Example1.prg`.

But what if run the program and find that you introduced bug? You will make some more changes, then do another conversion to .prg, but you'd get another error because your old, buggy .prg file is still there. You could delete the buggy `Example1.prg` and then re-issue the C64List command again. Better, you can just tell C64List to overwrite the buggy file. You can force C64List to overwrite the file by using the **-ovr** parameter.

```
C64List Example1.bas -prg -ovr
```

Now C64List will overwrite the buggy old Example1.prg with the new version. Of course, you will need to be judicious in your use of -ovr so you don't overwrite something important. Putting -ovr on a command line will cause C64List to overwrite *all* the output files you specified.

Converting to text: Cursor control characters

In the Quick Start section we saw how C64List converts the cursor control characters into handy, editable strings. If, for some reason, you don't want C64List to make that conversion, you can add -crsr to the command line to preserve the cursor control characters.

```
C64List Example1.prg -txt -crsr
```

```
100 REM THIS IS A BASIC PROGRAM FOR TESTING C64LIST
110 TW = 10 : HI$ = "□"
120 PRINT HI$ "THIS IS AN EXAMPLE BASIC PROGRAM"
130 PRINT "FOR TESTING"
140 PRINT "C64LIST!"
150 TW = TW -1
160 IF TW > 0 THEN GOTO 140
170 END
```

Although the graphics characters will appear as unintelligible ASCII characters when `-crsr` is specified, if left alone, will translate back correctly without issue when converting the file back to `.prg` format. Don't editing strings containing these characters; this will most likely destroy those characters. Unless you have a good reason to preserve the control characters, C64List recommends not to use the -crsr parameter.

Converting to text: Upper and lower case options

As you can see from the examples, when converting to the text format, C64List outputs BASIC keywords and variable names in uppercase. This preserves some of the nostalgic feel of programming directly on a C64. However, some people may find it easier to program on modern equipment using lowercase. C64List gives the option to output keywords and variable names in lowercase if desired. Adding -keycase to the command line will output all the BASIC keywords in lowercase. Likewise, adding -varcase to the

command line will output all the BASIC variable names in lowercase.

```
C64List Example1.prg -lbl –keycase -varcase
```

```
        rem THIS IS A BASIC PROGRAM FOR TESTING C64LIST
        tw = 10 : hi$ = "{black}{clear}{white}{down:4}"
        print hi$ "THIS IS AN EXAMPLE BASIC PROGRAM"
        print "FOR TESTING"
{:140}
        print "{right:10}C64LIST!"
        tw = tw -1
        if tw > 0 then goto {:140}
        end
```

When converting back to .prg format, C64List will automatically convert all keywords and variable to uppercase as appropriate.

It is also possible to change the case of quoted strings using the -alpha:lower command line option. See the section entitled *Mixed Character Case Support* for more information on alpha modes.

Supported input 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 can *read* the following file types:

Extension	File type
.prg	A tokenized BASIC, or machine language program.
.d64	A program file located inside a .d64 file
.txt	Plain-text code
.bas	Plain-text BASIC code
.lbl	Plain-text BASIC code, but without line numbers
.asm	Plain-text assembly code
.sym	Plain-text assembly code, used for symbol definitions

When reading a .prg or .d64 file, C64List automatically understands that it needs to detokenize the BASIC code into plain text. Typically, this operation won't be done very often—you'll do this once to get your code into editable format, then mostly convert the other direction.

The .txt, .bas, or .lbl file extensions signal that C64List expects plain-text source code files. These file types are interchangeable when reading—As input, C64List treats them in the same manner, and they may contain a mix of labeled or line-numbered code. They may contain BASIC code and directives, as well as assembly code enclosed in {asm} and {endasm} directives. Once the source code has been read, C64 can tokenize and/or assemble it into a .prg file that can be executed on a C64.

New behavior for C64List 4.00: C64List recognizes .asm files as assembly code. If you specify an .asm file

as your input on the command line, C64List 4.00 will enter casm (standalone C64List assembler) mode. This mode is specially designed for assembly-only source code, where no BASIC code is allowed. The casm tool is now integrated directly into C64List, so if you have used the casm tool in the past, this is new the way to invoke it. Please see the casm user's manual for more information on this mode. Further, C64List 4.00 automatically switches into casm mode when an .asm file is included or used from BASIC code; {asm} and {endasm} directives are not required around or inside .asm files.

If the file to be input has no extension, or if the extension is not what C64List expects, the -loadext:<ext> command line parameter overrides the file's extension so you don't have to rename the file. For example, the following command will load the file "Example1" (with no extension) as a .prg file.

```
C64List Example1 –loadext:prg -lbl
```

Supported output file types

C64List can *write* one or more of the output file formats it supports in a single run. Simply add the desired option to the command line and C64List will output a file of that type. C64List will write the following file types:

Parameter	Action
-prg	A tokenized BASIC or machine language program.
-bin	Same as prg, except the output file will not have a load address
-hex	An 8-bit hex dump of the program, as if it were loaded into C64 memory.
-txt	A detokenized, text version of a BASIC program, with various options.
-lbl	Same as txt, except with line numbers removed (see explanation below).
-lst	Machine language disassembly
-sym	Assembly language symbol table file

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 .bin format is identical to the .prg format, except that the load address is omitted. This is useful if you wish to generate a file containing only data, such as character definition data, for example.

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 with a monitor-type program on the C64 itself. C64List will write this format, but does not support loading files in this format.

As an output format, .txt files are plain-text, untokenized, line-numbered BASIC code. Note that the .bas format is not an output file type, since it is identical to .txt.

The .lbl format is similar to the .txt format, but uses labels instead of line numbers. See the following sections for more information on how to use this format. For loading purposes, this format is treated exactly the same as .txt files. When loading any of the .txt, .bas, or .lbl file formats, C64List will automatically identify numbered or numberless formatted code (or even a mix of both), and assign line numbers where needed. When outputting via the -lbl option, all line numbers are removed from the output and labels are inserted where necessary.

The .lst format outputs memory as a disassembled machine language program, by default, starting at the load address. Use the -range:$<startaddr>-<endaddr> option to limit output to a specific memory range. Note that any disassembler, including C64List, easily gets confused by inline data, and disassembly listing may become garbled. Attempting to disassemble BASIC code will also produce meaningless output.

Output file naming

C64List can write one or more of the output file formats it supports in a single run. Simply add the desired option to the command line and C64List will output a file of that type. If only the option is specified, the output file will take the name of the input file, and only the extension will be different. Optionally, the output file may be explicitly named. Here are some examples:

`C64list oldname.txt -prg`	the output file will be named `oldname.prg`
`C64list oldname.txt -prg:newname`	the output file will be named `newname.prg`
`C64list oldname.txt -prg:newname.bro`	the output file will be named `newname.bro`
`C64list oldname.txt -prg:newname.`	the output file will be named `newname`
`C64list oldname.txt -prg:.`	the output file will be named `oldname`

Output to console

The text-based output formats .txt, .lbl, .lst, and .hex may also be output to the screen instead of a file by specifying “con” as the filename. For example:

`C64list basicfile.prg -hex:con`	output is directed to the screen

D64 file format support

C64List supports reading and writing program files directly from and to .d64 files. Loading and saving files from .d64 is accomplished with very similar syntax as loading and saving in the Windows file system. The load filename and –prg:filename syntax has been expanded to allow for specifying a program file inside a .d64 file. The following example loads the file called `COMMODORE64FILE` from inside the D64 file called `D64File.d64` in the Windows filesystem, detokenizes it, re-tokenizes it, and then stores the resulting program in a file named `C64File.prg`.

```
C64List D64File.d64::COMMODORE64FILE –prg:C64File
```

Specifying the .d64 extension is optional. The double colon separates the D64 filename from the .prg file within the D64 file. It is important to note that while the D64 file's name is case insensitive, *the prg file within the D64 is strictly case sensitive*. Additionally, if spaces appear in the prg filename, you must enclose the filename in quotes.

Similarly, program files may also be written into to a D64 file using the same syntax:

```
C64List program.txt -d64:D64File::"C64 FILE"
```

In this example, a text-based source file is tokenized and then saved to a file called "C64 FILE" inside the D64 file called `D64File.d64`. For the output to go to a D64 file, you must use the -d64 command line parameter, followed by the D64::PRG formatted filename. This command line is an example of omitting the .d64 extension. Once again, the .prg filename is case sensitive and must be enclosed in quotes since there is a space in the filename. When saving to a D64 file, if the specified D64 file does not already exist, a new one will automatically be created for you. Otherwise, the file will be saved in the existing specified D64 file.

Adding add the -ovr command line parameter will *overwrite the entire .d64 file*, so take care with this parameter!

If you want to save-with-replace within the D64 file, use the C64 save-with-replace syntax:

```
C64List program.txt -d64:D64File::"@C64 FILE"
```

C64List Directives

C64List Directives are instructions to C64List to handle certain tasks or modify how some tasks are undertaken. To provide the most natural results, different classes of source code directives are defined, recognized and processed by four different layers of C64List, and handled during different phases of

C64List's work. Usually you won't need to think much about the differences, but understanding this concept gives some insight into the nuances of some of the directives. We'll discuss each type of directives in more detail.

- *Preprocessor Directives*: These directives control the loading and pre-parsing the source code, and are agnostic to whether the code is BASIC or assembly.
- *String Engine Directives*: These directives are only present within quoted strings, and add rich string-handling features to C64List. These directives work in both BASIC and assembly code strings.
- *Basic Tokenizer Directives*: These directives control and modify how BASIC code is processed; attempting to use these in assembly code will produce an error.
- *Assembler Directives*: These directives control how assembly code is handled; attempting to use these in BASIC code will produce an error.

Directives are not nestable; they are treated individually as they are encountered.

Preprocessor Directives

The first level of directives are handled by the preprocessor, which means that these directives apply equally to BASIC and assembly language source code. (note: This is different from C64List 3.xx, where directives were only allowed in BASIC.) A preprocessor directive resolves by being replaced by something in the source code at the time it is processed (some resolve to empty). This means that the directives themselves are never seen by either the BASIC tokenizer or assembler; the replaced text is what is passed on. If the preprocessor does not recognize a directive as its own, the directive get passed to the next level of processing (String Engine, BASIC, or Assembler). See *Appendix: Preprocessor Directives* for the full list of preprocessor directives.

String Engine Directives

C64List contains a module known as the C64List String Engine. This allows Windows to display coded C64-style strings (for example, C64 cursor code characters such as "{reverse:on}{red}{down:4}") as human readable codes. We already saw an example of this in the section *Converting to text: Cursor control characters*, above. Please see the section *Part 2: C64List String Engine* for more information on the String Engine itself, and *Appendix: String Engine Directives* for the full list of String Engine directives.

Basic Tokenizer Directives

The BASIC tokenizer directives, as their name suggests, are specific to BASIC code. A number of these directives are dedicated to labels and managing line numbers (e.g. {renumber}), while others are used to format source code (e.g. {keycase}) or control the output (e.g. {remremoval}). There are others as well. The BASIC directives are handed in various stages of the tokenization process: some apply to the entire BASIC code (for example, {loadaddr}), while others apply to specific areas of the code (e.g. {remremoval:on}/{remremoval:off}). The {alpha} directive appears in both the BASIC tokenizer and assembler. For the full list of this type of directives, see *Appendix: BASIC Tokenizer Directives*.

Assembler Directives

There are fewer assembler-specific directives, since most assembler controls are made through pseudo ops rather than directives. However there are a few. The {alpha} directive appears in both the assembler and BASIC Tokenizer. For the full list of Assembler directives see *Appendix: Assembler Directives*.

Including source code 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 uses the same third file, it eliminates a lot of file order dependencies

{uses} and {include} directives may provide a full path to the file. In this case, C64List will automatically add the location of the file to its search path list. You may also use the {addsearchpath: path1[, path*n*]} directive to add paths to the search path list. For {uses} and {include} that do not provide a full path, C64List searches for matching filenames in paths from the search path list.

Parser variables

Parser variables (aka preprocessor variables) can be defined to customize the C64List build process; for example, parser variables are used in **conditional compiling** (see the section on Conditional compiling for more information), or as a very simple text-replacement macro. Parser variables are not related in any way to BASIC variables or assembler symbols. They are only used by the source loader and preprocessor and are handled before any BASIC is tokenized or assembly code is assembled. These variables may be defined with the {def:<variable name>} directive. For example:

{def:compilethis}

Defining a parser variable automatically assigns a value of "" to the variable. You may specify your own string value to a parser variable using the following syntax:

{def:<variable name>=<value>}

Parser variables may also be defined from the C64List command line, using the –def:<variable name> command line parameter. Any number of –def parameters are allowed on the command line. For example:

```
C64List program.bas -prg -def:compilethis -def:skipthis
```

If desired, the value of the variable may be inserted into the source code using the {usedef:<variable name>} directive. The directive itself is replaced by the value of the variable in the source code prior to tokenizing or assembling. This type of text replacement is similar to a primitive form of macro.

Finally, a parser variable may be undefined with the {undef:<variable name>} directive. Undefining a variable completely removes it from the list of variables. For example:

{undef:skipthis}

System-defined parser variables

C64List automatically defines parser variables depending on how C64List is started:

- **__Casm** becomes defined when C64List is started in **casm** mode (standalone assembler mode)
- **__C64List** becomes defined when C64List is started in C64List mode (BASIC code)
- **__BuildRev** becomes defined when the **{buildrev:}** directive is used. The value is numeric and is incremented by 1 each time the project is built successfully.
- **__BuildDate** is set to the date C64List is run, in the form "YYYY-MMM-DD", enclosed in quotes.
- **__BuildTime** is set to the time C64List is run, in the form "HH:MM:SS", enclosed in quotes.
- **__BuildRid64** is set to a 64-bit random ID each time C64List is run. It is rendered as a string of 16 hexadecimal digits enclosed in quotes.
- **__BuildRid32** is set to the lower 32 bits of __BuildRid64. It is rendered as a string of 8 hexadecimal digits enclosed in quotes.

These parser variables may be useful to add to source files that might be used by both BASIC code and stand-alone assembler.

Conditional compiling

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

```
{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 parser 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} is not allowed.

In C64List 3.xx, these directives were only allowed in BASIC. In C64List 4.xx, they are now allowed anywhere in the code, BASIC or assembly. Additionally, as of version 4.00, multiple directives may be present on the same line of code, as well as mixed with BASIC or assembly code.

A conditionally excluded block of code must end with {endif}; C64List will inform you if you forgot to insert one.

Lines longer than 80 characters

Traditionally, Commodore 64 BASIC programs were limited to 80 characters. This is a limitation of the C64's BASIC editor, not the language or parser itself, so a .prg file containing lines with more than 80 characters will run fine. The caveat is that these long lines can't be modified on a real Commodore 64 or emulator. Attempts to modify such a line will result in the end of the line (beyond the 80th character) will be truncated.

C64List does not have a line length limit. You may enter and edit extremely long program lines if desired. Just remember you will not be able to edit your program using the C64's built-in editor. On the other hand, if you are certain that all your modifications will be made using C64List, then go ahead and make lines as long as you wish—programs will take up less memory that way!

“Crunched” program files

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 specify the -crunch parameter on the command line. When crunching is enabled, C64List will automatically remove all unnecessary spaces in your program before storing it in the destination file. Spaces inside quotes are left alone, since they are necessary to make your program output look right.

```
C64list Example1.prg -txt -crunch
```

```
100 REM THIS IS A BASIC PROGRAM FOR TESTING C64LIST
110 TW=10:HI$="{black}{clear}{white}{down:4}"
120 PRINTHI$"THIS IS AN EXAMPLE BASIC PROGRAM"
130 PRINT"FOR TESTING"
140 PRINT"{right:10}C64LIST!"
150 TW=TW-1
160 IFTW>0THENGOTO140
170 END
```

Here we can see that all the unnecessary spaces have been removed. You may notice that it is somewhat difficult to read. C64List can make your life much easier with its Autospace function when you are converting a tokenized file to text format. Autospace strategically inserts spaces into the output file to make it much more readable. Specifying the -autospace option when converting from a binary format file to a text format file (detokenizing) will activate autospace mode.

Here is the same example above, using the -autospace option:

```
C64list Example1.prg -txt -autospace
```

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

```
100 REM THIS IS A BASIC PROGRAM FOR TESTING C64LIST
110 TW=10 :HI$="{black}{clear}{white}{down:4}"
120 PRINT HI$"THIS IS AN EXAMPLE BASIC PROGRAM"
130 PRINT "FOR TESTING"
140 PRINT "{right:10}C64LIST!"
150 TW=TW-1
160 IF TW>0 THEN GOTO 140
170 END
```

REMark removal

Good programmers use lots of comment in their code. However, due to the limited amount of memory 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 was a REM statement, C64List will completely remove the line from the program. Once 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.

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. However, the modified code will behave the same when run.

Tick comments

C64List also supports a special type of comment that is not native to the C64. 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
150 DATA BUT THIS ONE : ' IS
```

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).

Tick comments are only valid in BASIC code; do not attempt to use this type of comment in assembly code.

Tick comments also don't apply to preprocessor directives—these directives are handled before BASIC ever sees the source code.

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. 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: • 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 *second* line of BASIC code after the directive. Restrictions: • 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 would naturally 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.

{:<*line number*>}	This is actually a label. However, numeric labels also serve as renumbering hints to the renumber function. C64List will attempt to re-assign the value of the numeric label as the line number. This is similar to the {number:<*line number*>} directive and is especially useful when converting from .prg to .lbl and then back to .prg.

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.

C64List will also accept source files where some line numbers are missing. The missing line numbers will be filled in automatically, if possible.

Convert to labels

Anyone who has ever programmed in Commodore BASIC has undoubtedly run into issues with line numbering. For example: you 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 easily convert it for you. Just specify the output file using the `-lbl` command line parameter. 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 were 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, such as {:Loop}. 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.

New in C64List 4.00:

1. numbered labels of the form {:<numeric value>} are called **keeper labels** and are used as hints when adding line numbers back into the .prg file. C64List will assign the line number inside the label to the next line of code, if possible.
2. When no renumbering is in effect, C64List will automatically attempt to fill in missing line numbers.

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 without using renumbering options would have created a faulty program.

Now that we have a nice, line-numberless, labelized 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.

Supervariables

C64 BASIC limits variable names to 2 characters. This can make BASIC source code cryptic and difficult to read. C64List has a feature called **Supervariables** to address this limitation and support long variable names.

A C64List supervariable is defined like this:

```
{var:LongVariableName=sh}
```

This tells C64List that the two-character BASIC variable **SH** is equivalent to the much more readable supervariable named LongVariableName. Using the newly defined LongVariableName in your code looks like these examples:

```
{var:LongVariableName} = 942
print {var:LongVariableName}
```

Here are some other working examples of Supervariables:

```
{var:superduperlongvariable=s0}
{var:SuperDuperLongString$=s1$}
{var:superduper%=s9%}

{var:superduperlongvariable} = 942
{var:SuperDuperLongString$} = "hello there!"
{var:superduper%} = -23280

print {var:superduperlongvariable}, {var:SuperDuperLongString$}, {var:superduper%}
```

When you run C64List on code containing Supervariables, C64List will output a summary of all the Supervariable assignments it found.

```
----------Supervariable assignments-----------------
{VAR:SUPERDUPER%=S9%}
{VAR:SUPERDUPERLONGSTRING$=S1$}
{VAR:SUPERDUPERLONGVARIABLE=S0}
----------------------------------------------------
3 Supervariable assignments found.
```

If you load the .prg file and LIST it on a C64, you will see that the above code compiled to this:

```
0 S0 = 942
1 S1$ = "HELLO THERE!"
2 S9% = -23280
3 PRINT S0, S1$, S9%
```

Converting an existing .prg to use Supervariables

C64List has a function for automatically converting an existing .prg file to Supervariable format. Use the -supervariables command line option to automatically detect variable names in your code, and automatically create long variable names for them. For example, we can convert the `supervariable.prg` file we just created back into a labelized listing as follows:

```
C64List supervariables.prg –lbl –supervariables
```

Opening the file, we see that C64List identified all 3 of the BASIC variables used in the program, assigned long Supervariable names for them, and then used them in the code as well:

```
{var:SuperVariable0000=S0}
{var:SuperVariable0001=S1$}
{var:SuperVariable0002=S9%}
{var:SuperVariable0000}= 942
{var:SuperVariable0001}= "HELLO THERE!"
{var:SuperVariable0002}= -23280
PRINT {var:SuperVariable0000}, {var:SuperVariable0001}, {var:SuperVariable0002}
```

Now that long variable names have been assigned, you can use your text editor's and search-and-replace function to replace the autogenerated names with something more meaningful.

Also note that this example shows that the variable type identifiers ($ and % for string and integer types) are optional in the Supervariable long names. For clarity, you may wish to include these symbols during your search-and-replace. The variable type identifiers are still required in the short names as usual.

Preface files

When converting from .prg format to text, there is no input source for adding directives, since directives go into text-formatted files, and not .prg files. C64List's **preface file** feature solves this by allowing a text-formatted preface file to be loaded immediately prior to the .prg file. In this way a user may add directives as if they were at the beginning of the .prg file.

Some usage examples for preface files is are:

- When the .prg file contains assembly code, the preface file can contain symbol assignments; symbol names will be inserted into the code where possible.
- Add {alpha}, {keycase}, {varcase} or other directives that customize the resulting outputted text.
- Define parser variables with {def} directives
- Define tokens when the .prg contains Custom Tokens to allow the resulting outputted text to contain the correct custom keywords

A preface file has exactly the same format as any other text-formatted C64List source file. There are some restrictions on preface files, however: The preface file cannot insert any bytes into memory. In other words, **preface files, and any files they may include must not contain any BASIC code, assembly code, data, or line numbers**.

To make C64List load a preface file, add the command line parater -pref:<pref_file>.

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 the program would not operate as one would expect. Your customized tokens must follow the same strategy, and take into account the already-defined tokens as well. Also, C64List supports only single-byte tokens.

New tokens may be specified in either upper or lower case; they are 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.

You may also use custom tokens when converting from .prg format to text. To do this, use a **preface 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. If you attempt to convert a .prg file that has tokens that C64List does not understand, and you have not provided a preface file to help out, C64List will simply output the hexadecimal value of the token enclosed within curly braces. C64List can also tokenize this format back into a .prg file. Here is an example that shows how C64List deals with custom tokens and unknown tokens.

Given the following program CustomTokensTest.bas:

```
{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 hypothetical weird extension of C64 BASIC would be able to run:

```
C64List CustomTokensTest.bas -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 CustomTokensTest.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. 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 text 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 the original `CustomTokensTest.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 preface file, as described in *Preface files*. The preface file is formatted exactly the same as the first few lines of our original text-formatted BASIC file. We'll call it `CustomTokens.bas`:

```
{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 CustomTokensTest.prg -pref:CustomTokens.bas -txt:GoodTokens -verbose
```

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

```
--------------------------------------------------------------------------------
                            C64List 4.00
                     Copyright (c) Jeff Hoag 2008-2019
--------------------------------------------------------------------------------
Starting C64List on CustomTokensTest.prg
Defining Parser Variable __C64List
Loading C:\proj\proj\c64listRework\TestFiles\CustomTokens.bas
[C:\TestFiles\CustomTokens.bas (1)] Modifying BASIC tokenizer
[C:\TestFiles\CustomTokens.bas (1)] Replacing token $e0 = GRAB
[C:\TestFiles\CustomTokens.bas (1)] Replacing token $e1 = EAT
[C:\TestFiles\CustomTokens.bas (1)] Replacing token $e2 = DROP
[C:\TestFiles\CustomTokens.bas (2)] Modifying BASIC tokenizer
[C:\TestFiles\CustomTokens.bas (2)] Replacing token $e3 = SLEEP
====================
0 Errors; 0 Warnings
====================
C64List finished
```

Here, we can see that C64List found our preface 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, you may also undefine existing tokens. In the {tokenizer} directive, simply specify the token you wish to undefined, 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.

Load address

If, for some reason, your program needs to load to an address other than the default address 2049 ($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:con
```

```
0801: 1a 08 0a 00 99 20 22 48 45 4c 4c 4f 20 43 36 34|..... "HELLO C64
0811: 20 55 53 45 52 53 21 22 00 20 08 14 00 80 00 00| USERS!". ......
0821: 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
```

```
5001: 1a 50 0a 00 99 20 22 48 45 4c 4c 4f 20 43 36 34|.P... "HELLO C64
5011: 20 55 53 45 52 53 21 22 00 20 50 14 00 80 00 00| USERS!". P.....
5021: 00                                             |.
```

And voilà! We see that we successfully changed where the program loads to 20481 ($5001). 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).

When you convert a .prg file containing BASIC code that has a non-standard load address (anything but $0801) to a text format, C64List will automatically insert a {loadaddr} directive at the beginning of the file to preserve the original load address.

Note: {loadaddr} is a BASIC directive and can't be used in assembly-only projects. For those, please use the orig pseudo-op.

Build revision tracker

C64List can track incremental build revisions automatically for you. To use this function you'll need to do a few things:

1) Create a build revision file and add a single ASCII decimal value into it. For example you'd likely put a single 1 digit in the file. No spaces, carriage returns, or any other text or characters are allowed anywhere in the file.
2) Use the {buildrev:<*filename*>} directive in your source code to tell C64List where the build revision file is located and that you want to track build revisions. The value is numeric and must be treated as such.

3) Insert the build revision somewhere in your project using the automatic parser variable named __BuildRev in a {usedef:__BuildRev} directive.

Now, whenever C64List successfully builds your project with no errors, the build rev value will be stamped into your project and then the build rev ID in the file will automatically be incremented by one for the next build. Each of your projects that utilize this feature will need to have its own build rev file. The build rev file can be committed to a source repository along with the rest of your source code if desired. Please ensure that the file is writable before attempting to build the project. It is possible for the build rev ID to grow to a very large number, beyond the capabilities of a mere 16 bits. If the ID exceeds 65535, your code may generate an error. It's a good practice to manually edit the build rev file to reset it to a lower value any time you update the program version number. Here's an example program that uses this feature twice: once in BASIC and once in assembly code.

Created a file called BuildRev.rev with the text **1** in it.

```
100
```

Created a file called BuildRevExample.bas with the following contents:

```
{buildrev:buildrev.rev}
10 print "example program version 1.00, build revision" {usedef:__BuildRev}

{asm}
VersionMajor: byte 01
VersionMinor: byte 00
BuildRevision: word {usedef:__BuildRev}
{endasm}
```

Part 2: C64List String Engine

C64List was created to allow software developers to develop C64 code in the Windows editor of their choice. One side effect of this is that most Windows text editors are not equipped with the C64 character sets, and thus can't display all the characters available in the traditional C64 on-screen editor. This means that the already cryptic cursor control characters that appear within quoted strings would appear essentially un-readable and un-editable in a Windows editor. C64List handles this elegantly by outputting the cryptic cursor control codes into human-readable directives such as {blue} and {clear}, and then returning them to the C64's native cursor control codes when the source code is tokenized into a .prg file. The piece of C64List that handles these conversions is known as the C64List String Engine.

The C64List String Engine knows how to handle the C64 cursor control codes, color controls, reverse controls, upper and lowercase, unprintable characters, and graphical glyphs to aid the cross-platform programmer. It also handles repeated characters in a shorthand form such, for example {down:10}, meaning ten consecutive cursor down controls.

The String Engine works for strings in BASIC source code, as well as strings and character constants in the assembler. String Engine directives are active only inside the quotes of strings, similarly to the way they work natively on the C64. Please see the full list of String Engine Directives in the section *Appendix: String Engine Directives*.

The behavior of String Engine may be controlled with other directives that belong to the BASIC tokenizer and assembler. These control directives tell the String Engine how the program intends to use the strings, for example, using the standard character set or the alternate (mixed-case) character set. These settings are discussed below, and may also be found in the full list of BASIC directives as shown in the section *Appendix: BASIC Tokenizer Directives*.

Mixed character 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. This discussion is about printable strings enclosed in quotes.

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 character case support* (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.

Advanced topic: It's rather confusing so if you don't need to know about it, simply skip this paragraph and avoid using the **invert** options listed below. For those who are interested, PETSCII's alternate character set defines lowercase characters starting at $41, and uppercase starting at $C1. This is in contrast to ASCII, which defines uppercase characters starting at $41, and lowercase starting at $61. To make matters more confusing, PETSCII also defines duplicate uppercase characters starting at $61. So although code may appear correct with these values, the program may not operate correctly. Even more strange is that C64 does not render alternate character set uppercase correctly in REMark statements. Instead of uppercase letters, C64 will detokenize BASIC keywords. Therefore, uppercase REMark

statements must use the duplicate uppercase characters in the $61 range. C64List handles all this confusion automatically when you use the alpha options listed below.

When converting *from* .prg files:

C64List has no way to know whether the characters in the .prg file are intended for the normal or alternate character sets. However, you can ask C64List to do a conversion for you by including the –alpha command line option. This feature will convert text that is enclosed within quote marks and REMark statements. The following table lists the available options. In the table, [std] indicates that the option is intended for programs that use the standard character set, while [alt] indicates that the option is intended for programs the make use of the mixed-case alternate character set.

-alpha:ascii	Don't do any conversion; leave strings in ASCII coding
-alpha:upper	[std] Convert character strings in tokenized BASIC to uppercase ASCII
-alpha:lower	[std] Convert character strings in tokenized BASIC to lowercase ASCII
-alpha:normal	[std] Same as -alpha:upper
-alpha:alt	[alt] Correct the characters' case in tokenized BASIC to ASCII
-alpha:invert	Invert the upper/lowercase ASCII values (deprecated; not recommended)

*Normal is the default option

If you don't specify any –alpha command line option, C64List will assume that the program uses the standard character set and ensures that all quoted characters and REMark statements are output in uppercase. The **upper** option is exactly the same. The **lower** option is the similar, except that the characters are output in lowercase. These options are simply for your preference.

The **alt** option tells C64List that the program uses the alternate character set with upper and lowercase characters. In this case, quoted and REMark characters are converted to upper and lowercase so they look correct in the output .txt or .lbl files.

Typically, you will not want to use the **invert** option. In older versions of C64List, this was the only option for handling the alternate character set, so this option is provided only for backward compatibility.

Finally, the **ascii** option will prevent C64List from making any alpha conversion. Uppercase and lowercase characters in the tokenized BASIC code will remain in ASCII encoding and appear to be swapped in your .txt or .lbl files.

When converting *to* .prg files:

C64List can also make alpha conversions when converting from text-based source code to .prg files. Embed alpha directives into the source code at key locations to tell C64List how to handle characters. You may notice that some of the {alpha} directive options have the same name as the command line

options. Despite this fact, most of the options have different meanings when converting in this direction. The following table lists the options available when converting text-based source code to tokenized BASIC .prg files. [std] and [alt] indicate usage for the standard and alternate character sets.

{alpha:ascii}	Don't do any conversion
{alpha:normal}	Same as {alpha:lazy}
{alpha:lazy}	[std] Converts all ASCII upper and lowercase letters to uppercase *in the standard character set*. ($41 range)
{alpha:alt}	[alt] Corrects the upper and lowercase letters' case in the program *in the alternate character set*.
{alpha:poke}	[std] Convert letters and symbols for use when POKEing characters to the screen *in the standard character set*. This mode is similar in function to lazy, as it converts both upper and lowercase ASCII characters to uppercase POKE codes.
{alpha:pokealt}	[alt] Convert letters and symbols for use when POKEing characters to the screen *in the alternate character set*.
{alpha:upper}	[alt] Convert all letters in the source code to PETSCII uppercase in the program *in the alternate character set*. ($C1 range)
{alpha:lower}	[alt] Convert all letters in the source code to PETSCII lowercase in the program *in the alternate character set*. ($41 range)
{alpha:invert}	Invert the upper/lowercase ASCII values (deprecated; not recommended)

*Normal is the default option

For the standard character set, the simplest option to use is **{alpha:lazy}** (which is the same as **{alpha:normal}**, or not specifying anything since it is the default). This directive allows you to type your source code in upper and lower case, and automatically converts all letters to uppercase in the standard character set. This is the default, so you don't need to explicitly specify it.

To indicate that a program is meant for mixed-case on the C64, place the **{alpha:alt}** directive in your code. This will cause the case of all alpha characters to be corrected, so they show up correctly when listed on the C64 in the alternate character set.

The **{alpha:poke}** directive is useful when you want to poke characters on the screen in machine language. Since the screen codes are different from the printable character codes, POKEing the letter 'A' to the screen requires you to use $01 instead of the normal printable $41. This directive allows you to define a POKEable string that is also readable in source code. Type the string in ASCII and C64List will convert the characters to the POKE screen codes automatically.

The **{alpha:upper}** and **{alpha:lower)** options are for use with the alternate character set, and convert all letters to uppercase or lowercase, respectively. These options will probably not get much use.

The **{alpha:invert}** is, like the command line option, not too useful, but provided for backward compatibility.

Finally, to turn off the character conversion function, use the directive **{alpha:ascii}**. ASCII values of the characters you typed will be passed directly into the .prg file. Uppercase characters in the source code will appear as lower case in the tokenized BASIC, and vice versa.

You can change the alpha handling mode wherever you wish, and however often you want, in your code.

Graphics glyph support

C64 has cool graphic glyphs that do not translate to ASCII. This makes such graphics impossible to visualize in a Windows console. Therefore, C64List handles quoted graphics characters by displaying the code as a numeric token enclosed on curly braces. For example, 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:alt}** and then it will come through as an uppercase A.

Similarly, when you convert from .txt to .prg, C64List will convert any such quoted numerical tokens of the form {$61} to the actual character code. 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 yet 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{lift}. 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 glyph token names

As an extension of the cursor control code feature, C64List supports custom glyph token names. You may add, modify and remove token names using the {quoter} directive. For example, if you would like to replace the shifted A (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.

Part 3: Assembly code

C64List not only can detokenize and tokenize C64 BASIC program, it also contains an integrated 6510 assembler and disassembler. C64List 3.00 had a pretty good assembler and disassembler, but it had been added on top of the BASIC handling code and was difficult to maintain. So the project was temporarily branched into a stand-alone assembler called **casm**, to develop and mature the code base. C64List 4.00 recombines the C64List BASIC functionality with the 6510 assembler functionality into a single tool. In this section we will discuss their features and how to use them in your C64 code development.

C64List hex dump utility

When developing or debugging assembly code, it is often useful to dump the program in hex format. C64List will do this for you if you add the command line parameter **-hex:<file>**. We have already used this feature earlier in this document. As with other text-formatted output, you may dump directly to the console instead of a file by using the name **con**; for example **-hex:con**.

As an example, let's create and look at a file called Mixed.prg. Here's the source code file, Mixed.bas that we can convert to .prg:

```
sys {sym: AssemblyFunction}

{asm}
Screen = 1024
Checker = $ff

AssemblyFunction:
       lda #Checker
       ldx #$27
L0:
       sta Screen, x
       dex
       bpl L0
       rts
{endasm}
```

Now let's convert to .prg so we can examine it.

```
C64List Mixed.bas -prg
```

Now that we have a .prg file, let's look at a hex dump of it:

```
C64List Mixed.prg -hex:con
```

```
--------------------------------------------------------------------------------
                            C64List 4.00 Beta B
                      Copyright (c) Jeff Hoag 2008-2019
--------------------------------------------------------------------------------
Starting C64List on Mixed.prg
0801: 0d 08 00 00 9e 20 20 32 30 36 33 00 00 00 a9 ff|.....  2063.....
0811: a2 27 9d 00 04 ca 10 fa 60                     |.'......`
```

Recall that the first two bytes in a .prg file indicate the memory address within the C64 where the program is to load. The first two bytes of the file are extracted from the data and interpreted as the address memory. As you can see here, the leftmost column contains the memory address $0801, which is the default load address for BASIC code in the C64. The address is followed by a colon (":"), and then the contents of up to 16 bytes of memory. These bytes are then followed by a vertical bar ("|") and 16 characters that show the ASCII equivalent of the previous 16 bytes. Unprintable characters are shown as a period ("."). We'll look more at this file in the following sections.

If the file is large, it may be useful to only dump part of the file. To do this, use the **-range** command line parameter. The range parameter's syntax is **-range:<start>** or **-range:<start>-<end>**. The starting and ending addresses may be given in hex (using the dollar sign prefix "$"), or decimal (no prefix). If no ending address is given, the end-of-file is used.

Too look at just the last line of Mixed.prg, we'll hex dump a file to the console, starting at address $0811:

```
C64List Mixed.prg -hex:con –range:$0811
```

```
--------------------------------------------------------------------------------
                            C64List 4.00 Beta B
                      Copyright (c) Jeff Hoag 2008-2019
--------------------------------------------------------------------------------
Starting C64List on Mixed.prg
0811: a2 27 9d 00 04 ca 10 fa 60                    |.'......`
```

In this case we didn't specify any ending address, so C64List showed us from $0811 to the end of the file.

We can narrow the output even further if we specify an ending address:

```
C64List Mixed.prg -hex:con –range:$0811-$0814
```

```
--------------------------------------------------------------------------------
                            C64List 4.00 Beta B
                      Copyright (c) Jeff Hoag 2008-2019
--------------------------------------------------------------------------------
Starting C64List on Mixed.prg
0811: a2 27 9d 00                                   |.'..
```

As requested, C64List output only 4 bytes.

C64List disassembler

We saw the hex dump of the file Mixed.prg in the previous section. Now let's look at the BASIC code:

```
C64List Mixed.prg -txt:con
```

```
--------------------------------------------------------------------------------
                            C64List 4.00 Beta B
                      Copyright (c) Jeff Hoag 2008-2019
--------------------------------------------------------------------------------
Starting C64List on Mixed.prg
0 SYS  2063
 {warning:Source .prg file contains non-BASIC code which will need to be ported
separately: $080f 0819}
```

We can see that the first few bytes of the file are indeed BASIC code, and consist of a SYS call to an address somewhere in memory. Referring back to the hex dump, we see that 2063 ($080f) is the address immediately following the end-of-BASIC marker. This is the method C64List uses to pack BASIC and assembly code in the same file: all BASIC source code is tokenized and put into memory first, then assembly code is assembled starting at the next address after the end-of-BASIC marker.

C64List also automatically inserted a warning into the output BASIC source code telling us it found some additional bytes after the end-of-BASIC mark. The bytes are not part of the BASIC program itself, so we'll need to do some more investigation to find out what those bytes are. This is where C64List's disassembler comes in handy. This C64List feature allows machine-language .prg files to be disassembled into human-readable assembly instructions. To invoke the disassembler, simply add the **-lst** output parameter. We'll disassemble to the screen as we did with the hex dump.

```
C64List Mixed.prg -lst:con
```

```
--------------------------------------------------------------------------------
                            C64List 4.00 Beta B
                      Copyright (c) Jeff Hoag 2008-2019
--------------------------------------------------------------------------------
Starting C64List on Mixed.prg
        080f  a9 ff      lda #$ff
        0811  a2 27      ldx #$27
        0813  9d 00 04   sta $0400,x
        0816  ca         dex
        0817  10 fa      bpl $0813
        0819  60         rts
```

As you can see, the disassembly output has several columns. The address comes first, followed by the opcodes of the instruction at that address. A single 6510 instruction may consist anywhere between 1 and 3 bytes. These opcode bytes are shown after the address. C64List then interprets the opcode bytes and outputs the assembly instruction and its operands in the next two columns.

Our clue that this data is actually machine code was that the BASIC part of the file contained a SYS to the addresses after the end-of-BASIC marker (2063, which is equivalent to $080f). Even though we didn't specify where to start disassembling, the first address in this listing is $080f, not the load address of $0801. That is because C64List scans past any BASIC code in the file and then begins disassembly immediately after the end-of-BASIC marker. As with the hex dump above, the listing continues to the end of the file.

In this case, it appears that the data after the end-of-BASIC marker is indeed valid machine code. It is a small loop that writes some characters to top of the screen and then returns to BASIC. However, any disassembler will simply attempt decode bytes as if they were valid assembly instructions. Therefore, if you attempt to disassemble memory that does not contain a machine language program, the result will be meaningless. For fun, we can ask C64List to attempt to disassemble the BASIC part of the file.

```
c64list Mixed.prg -lst:con -range:$0801-$080e
```

```
Starting C64List on Mixed.prg
        0801  0d 08 00   ora $0008
        0804  00         brk
        0805  9e 20 20   ??? $2020,y
        0808  32         ???
        0809  30 36      bmi $0841
        080b  33 00      ??? ($00),y
        080d  00         brk
        080e  00         brk
```

Since we forced C64List to try and interpret the BASIC program as machine code, C64List blindly followed our instructions and, and as expected we ended up with meaningless junk!

Inline data within machine code can also cause the disassembly listing to get off track, since a disassembler has no way of identifying data versus code. The point of this is that it will take quite a bit of investigative work to correctly disassemble machine language into readable assembly code.

Another tool that can help to digest unfamiliar code and make it more readable, is C64List's label-stuffing feature using a **preface file**. Add a **-pref** command line option and specify a file that contains symbol assignments. As you identify subroutines and other interesting addresses, you can name them by adding symbol assignments to the preface file.

Now let's create a pref file called Mixed.sym. Normally, you will need to investigate to determine what symbols to define. But for our example, we'll cheat a little—since we started the example with a source file, we already know some symbols. So let's add into Mixed.sym:

```
Screen = 1024
Checker = $ff
Loop = $813
```

Now we can disassemble Mixed.prg with the symbols we assigned.

```
C64List Mixed.prg -lst:con –pref:Mixed.sym
```

This time we get a listing that is a little more readable! The loop is labeled **loop**, and the bpl instruction also refers to the label instead of address $0813 as before. Also the sta instruction's operand uses the symbol **screen** instead of the address 1024.

```
Starting C64List on Mixed.prg
        080f  a9 ff      lda #$ff
        0811  a2 27      ldx #$27
loop:
        0813  9d 00 04   sta screen,x
        0816  ca         dex
        0817  10 fa      bpl l0
        0819  60         rts
```

You may have noticed that although we added the symbol **Checker** to the pref file, the lda #$ff instruction did not use the symbol. That is because C64List only replaces *address operands*, and not immediate values. Why? Because assigning names to small values can easily create misleading code. For example: Although it might seem to make sense to assign the name "CarriageReturn" to the value $0D, remember that the disassembler is not able to distinguish between usage, and would have replaced any instance of the value $0D with the name "CarriageReturn". So while "lda #CarriageReturn" may make sense, loading a value from the address $000D (i.e. "lda $0D") would have showed up as "lda CarriageReturn" and would not be an accurate description of what is happening. So C64List avoids doing this.

C64List's disassembler is aware of pointers. Therefore, even if an operand is accessing one past a known symbol, C64List will reference the symbol +1 to try and make the disassembly code more readable. Occasionally, this may not be show what you would expect (even though it is still functionally correct), so just be aware of this feature.

C64List symbolic assembler

The new C64List assembler has two modes: **C64List mode** and **casm mode**. While the two modes are pretty similar and share the same code, there a few differences.

C64List assembler mode
C64List mode allows BASIC code and assembly language code to be combined into a single .prg file. Launching into the assembler in C64List mode is no different from starting C64List to tokenize a BASIC program. C64List assumes that all source code in the input file is BASIC until it finds an {asm} directive, at which point it transitions into assembler mode. Likewise, an {endasm} directive exits the assembler and returns to BASIC. There can be any number of {asm} / {endasm} pairs in your code, and they may be mixed anywhere into the BASIC code. Additionally, including any file with the .asm or .sym extenstion using {include} or {uses} automatically switches to assembler mode for the duration of the file, and no {asm} / {endasm} pair is required in this case.

C64List collects all of the assembly code pieces, locates the end-of-BASIC marker, and places all the machine language *after the end of the BASIC code in memory*, immediately after BASIC's final NULL link.

Some rules for C64List assembler mode

- All assembly-language instructions in BASIC type files (.bas/.txt/.lbl) must be placed inside {asm} and {endasm} directives.

- There can be any number of {asm} and {endasm} directive pairs
- Assembly code may be placed anywhere in the BASIC code; C64List will automatically collect it all and locate the machine code after the end of BASIC.
- Assembly files (.asm/.sym) may be included from BASIC type files without any use of {asm} and {endasm} directives.
- Assembly files (.asm/.sym) may contain only assembly code; BASIC is not allowed.
- Assembly files (.asm/.sym) may not include any BASIC type files

casm mode

Launching C64List in casm mode is done simply by specifying an .asm file as the input filename. In casm mode, C64List expects *all* source code to be assembly language; further, it is not possible to switch into BASIC either with {endasm} nor by {include}'ing, {uses}'ing .bas or .txt files. {asm} and {endasm} directives are ignored in casm mode, and C64List will report an error message and exit if an attempt is made to include BASIC-type files (.bas, .txt, and .lbl). Casm mode exists to simplify assembly-only projects. It contains all the features of C64List mode, plus some additional features that are not available in regular C64List mode--due to the fact that those features do not make sense when any BASIC is involved. If you plan to develop a program exclusively in assembly language, casm mode is recommended.

Files that contain only assembly code should be named with the **.asm** extension, and assembly code in these files do *not* need to be enclosed in {asm} and {endasm} directives. This allows C64List to be invoked in casm mode simply by C64List-ing the file. It also allow any reusable .asm files to be seamlessly included from other BASIC code from C64List mode, if desired.

As of C64List 4.00, some casm-only features are still under development and not available for use. Therefore, there's currently not a whole lot of difference between C64List mode and casm mode, except for the convenience factor of simply invoking C64List with an .asm file and the lack of {asm}/{endasm} directives. More information will be added to this document once casm mode features have been released. Assembler features described in this document are available in both casm mode and C64List mode, unless otherwise noted.

Some rules for casm mode

- Assembly files (.asm/.sym) may contain only assembly code; BASIC is not allowed.
- Assembly files (.asm/.sym) may not include any BASIC type files
- The {asm} / {endasm} directives are optional in .asm and .sym files. Their use is provided for backward compatibility with C64List 3.xx, but discouraged.

General assembler syntax rules

- Comments are identified and preceded by a semi-colon (";")
- Everything after a semi-colon is considered to be a comment
- Tick-type BASIC-style comments ("'")are not allowed in assembly code
- Source Loader directives *are* allowed in assembly code
- String Engine directives *are* allowed in assembly code

- BASIC directives are *not* allowed in assembly code
- Only a single assembly instruction or pseudo op is allowed per line of code
- Four assembly language elements are allowed per line of code--one of each, some of which are optional:
    - A label (optional)
    - An instruction
    - An instruction operand (required, if the instruction requires one)
    - A comment (optional)
- Implied addressing mode instructions, such as the rotate and shift accumulator instructions, are not allowed an "a" operand (strangely, some other 6510 assemblers require one).

Code Parsing

Similarly to C64List BASIC tokenization, the assembler makes several passes when assembling.

1. The first is the preprocessor pass. The preprocessor is the same for both BASIC code and assembly code. One of its jobs is to separate the lines of assembly code from the lines of BASIC code. In the case of casm, of course, there can be no BASIC code. The preprocessor directives (see below) are interpreted at this time, and text-replaced with the desired values, as appropriate. For example, {include:<file>} is replaced by the contents of the file, and {$:10} is replaced by "16".
2. Comments are removed from the assembly code line
3. Lines of assembly code are scanned to find what type of line they are, and what code elements exist.
4. The code elements are assembled to machine language opcodes, or a pseudo op action is performed, as necessary
5. Instruction and pseudo op operands are evaluated
6. The program is "linked"; memory addresses are assigned to each label and instruction
7. Operands are evaluated again to fill in recently assigned label values
8. Opcodes are inserted into memory at the correct addresses

There are two classes of lines in assembly, **equates lines** and **code lines**. Equates lines are lines of assembly that declare a symbol, while code lines contain assembly instructions or pseudo ops.

Symbols

Symbols are essentially assembler constants. As they are defined, the assembler collects them into a Symbol Table where they can be referenced by the source code at a later time. Symbols may be declared in an equates line by naming the symbol and assigning a value using an expression. The expression may consist of numeric or character constants, other symbols, or an expression. Symbols must resolve into a single 16-bit value before the assembly code can use them. C64List attempts to resolve symbols at various times throughout the code assembly process, so the symbol definition is not required to be physically precede its usage in source code.

The assembler looks for the equal sign to determine if the line is an Equates line. Examples of Equates lines:

```
Symbol1 = $5000                 ;a constant numeric value
Symbol2 = *                     ;the value of the current address
Symbol3 = Symbol1 + Symbol2     ;using an algebraic formula
Symbol4 = 'x'                   ;a constant character value
Symbol5 = '='                   ;equal sign inside quotes is legal
```

Rules for symbols

- Assembly symbols are completely separate from BASIC line labels, and BASIC variables, and parser variables.
- Symbols are case insensitive
- Symbol names must start with a letter and may contain numbers and an underscore
- A symbol is defined in an equates line of the format <*symbol*> = <*expression*> like this:
  ```
  Screen = 1024
  ```
- Symbols may only be defined once; any attempt at redefinition will produce an error
- Constant values in an expression may be notated in hexadecimal (prefixed by a dollar sign "$"), binary (prefixed by a percent sign "%"), or decimal (no prefix)
- A symbol must completely resolve into an 8- or 16-bit value, or an error will be generated
- Circular references will generate an error; for example:
  ```
  Address = Address + 1
  ```
- A symbol may be referenced either before or after it is defined in the code

Labels

Labels allow coders to code a jump instruction to some other instruction in the program without having to know its address. A label is actually just a symbol, and is stored in the symbol table like any other symbol. However, labels specifically refer to an address somewhere in the source code, they are defined as part of an instruction on a code line (rather than in an equates line), and their value is automatically assigned by the assembler during the link phase. See the examples in *Code lines* below.

A label can be declared on a line with code, or it can be declared on a line of its own, with no code. In the second case, the label will be assigned the address of the next byte that is assembled to memory. Because of this, it is possible to declare multiple labels for the same line of code. This ability may be useful at times, for example when a local label and general label both refer to the same location.

Assembly functions should be labeled to allow them to be called from BASIC or other assembly routines.

Rules for labels

- An assembly label is simply a symbol whose value is automatically set to the address of a line of code, and is added into the symbol table with all the other symbols. Therefore, all the symbol rules also apply to labels.
- To define a label, pick a symbol name that has not yet been defined and follow it with a colon

```
Label: 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. They will all be assigned the same value when the code is linked.

```
Label3:
Label4:
Label5: lda #$ff
```

Symbol levels

Large assembly language programs tend to contain a lot of symbols. In most cases, you won't care about the exact value of every symbol, and finding the important ones can be difficult in a large list of symbols. C64List has several closely-related features to help manage symbols: **symbol levels**, **local labels**, **anonymous labels**, and **symbol demotion**.

The symbol levels feature helps filter out less interesting symbols when the symbol table is exported. C64List assigns an importance level to every symbol, and then when the symbol table is dumped, by default only the most important symbols are shown. There are three importance levels:

General	All symbols are assigned the level general unless specifically requested.
Important	Important symbols might include externally visible labels, such as main function names
Local	Used for loops and jumps, when only the current function cares what the value is

General symbols are defined as usual; either directly in an Equates line, or as a Label in a Code line. Important symbols are defined in the same way, except they are preceded by an exclamation mark.

```
GeneralSymbol = 18
!ImportantSymbol = 9
!ImportantLabel:          lda #GeneralSymbol
GeneralLabel:             ldx #ImportantSymbol
                          rts
```

Notice that the exclamation mark is not part of the symbol's name; it is only used when *defining* the symbol or label, *not when referencing it*. When you compile this code snippet and output the symbol table, you will see that only the important symbol and important label are listed; the lesser important ones were suppressed. By default, C64List finds the highest level of symbol that is defined and only dumps symbols at that level.

```
C64List ex.txt –sym:con
```

```
------Symbol Table------
importantlabel   = $0100
importantsymbol  =   $09
------------------------
```

If you want to see the general symbols in the symbol table too, you may modify the command to tell C64List to show the general symbols as well:

```
C64List ex.txt –sym:con –symlvl:general
```

```
------Symbol Table------
importantlabel    = $0100
importantsymbol   =   $09
generallabel      = $0102
generalsymbol     =   $12
------------------------
```

Local labels

Local labels can be thought of as throw-away labels, and are the least important level of symbol. Many times you will want to loop or skip some instructions; in this case, you only need the label internally to the code, and don't care what the actual value is. Additionally, you don't really care what the name is, and you'd prefer to keep it short. Without local labels, you can't use simple label names like "loop" because they would likely be defined elsewhere as well and you would easily generate an assembler error. This is particularly important when using modular programming methods and code re-use.

With local labels, you may define the same label name as many times as you wish; references to these labels refer to the *nearest* instance of the given label in the indicated direction. Local labels are defined with the ***at* symbol** ("@"). In contrast with important labels, local labels *must be referenced* using the @ symbol as well, in addition to the *direction* in which the label is to be found. The *greater than* symbol (>) indicates a forward search direction, while the *less than* symbol (<) indicates a reverse search direction.

```
Screen0 = 1024
@label:      brk
@label:      lda screen0
             bne <@label
             beq >@label
@exit:       brk
```

```
@label:      rts
```

As you can see in the above example, the local label @label was defined 3 different times! However, the assembled code shows that it compiled as desired. The bne jumps back to the previous label, and the beq jumps forward to the rts.

```
        0100  00         brk
        0101  ad 00 04   lda $0400
        0104  d0 fb      bne $0101
        0106  f0 01      beq $0109
        0108  00         brk
        0109  60         rts
```

If, for any reason you wish to see the values assigned to the local labels, you may ask C64List to show you all the labels in the symbol table. The following symbol table is generated from the first local label snippet above. As you can see, C64List automatically and silently adds decorations to the local label names to ensure they aren't confused internally.

```
C64List ex.txt - -sym:con –syml:all
```

```
--Symbol Table---
@exit@0   = $0108
@label@0  = $0100
@label@1  = $0101
@label@2  = $0109
screen0   = $0400
-----------------
```

Anonymous labels

If you really don't care about the name of a label, you may use an anonymous label—unnamed, and comprised only of the @ symbol. As with other local labels, you may use an anonymous label as many times as needed. This is really helpful when you have a series of short jump-overs, similar to an if-elseif-elseif- sort of situation. This is nice because you are no longer forced to think up mostly meaningless names for each label. For example:

```
Screen0 = 1024
@:           lda screen0
             cmp #'a'
             beq >@
             cmp #'A'
             beq >@
             jmp <@
@:           rts
```

Symbol demotion

When developing a re-usable code module, it is often useful to define important-level symbols or labels. However, once the code is finished and debugged and being used in another project, the important symbols in the module typically become less interesting to the new project as a whole. To remedy this, one might edit the module and remove the important declarations to make them general so they won't clog up your newer, more important symbols. C64List has a better way that doesn't require editing the original module. You may selectively turn on symbol demotion when including the module with the undesired important symbols, using the {symdemote} directive. For example:

```
{symdemote:on}
{uses:usefulmodule1.asm}
{uses:usefulmodule2.asm}
{symdemote:off}
{uses:usefulmodule3.asm}
!Entry:
```

Now all of the important-level symbols defined in usefulmodule1.asm and usefulmodule2.asm will be demoted to general-level symbols. However, any important-level symbols defined in usefulmodule3.asm will retain their importance in this example, since we turned off symbol demotion before we included the third file.

Operands

Symbol definitions, instructions, and pseudo ops may require one or more operand. The exact format of an operand is dependent on what it is attached to. Most operands are numeric, in which case it may be an **expression**. Some pseudo ops (such as byte and word) allow multiple operands, separated by commas. Other pseudo ops require a string operand, and others can accept a mixture of numeric expressions and strings.

String operands in assembly are processed through the C64 String Engine in much the same way as strings are handled in BASIC code.

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
%10010001
```

Examples of non-zero page addresses. Notice how leading zeros are used to eliminate ambiguity in the last two values:

```
$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
nZp2 = Zp+nZp;nZp2 is an absolute 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
```

In cases where zero page values are never allowed, C64List automatically updates the operand to non-zero page. For example the following should cause an error:

```
jsr $45
```

However, C64List will automatically convert it to a 16-bit value and assemble without error:

```
jsr $0045
```

Expression Evaluation

A significant change from the 3.xx C64List versions is that numeric operands of instructions and pseudo ops are now all treated as **expressions**, and expression evaluation has been greatly improved. Unlike C64List 3.xx, expressions are now processed with arithmetic operator precedence. Because of this, you may need to change some of your older 3.xx assembly code. For example, if you have code that contains an expression using both addition and multiplication, C64List 4.00 may interpret it differently from C64List 3.xx. Take, for example, the following code:

```
        lda #<start + 50 * 2
```

In 3.xx, the operand would have been assembled strictly from left-to-right and would have been interpreted as

```
        ((start + 50) * 2)
```

C64List 4.00 improved the situation so that the commonly-accepted arithmetic precedence rules are followed, and the same expression would be interpreted more correctly as

```
        start + (50 * 2)
```

C64List provides a number of elements that may be used in expressions:

Number	A numeric constant, which can be expressed as $hex, %binary, or decimal
Symbol	A symbol or label defined somewhere in the code
Special Symbol	A C64List-defined symbol, such as * or ?
Character	A single character or C64List-style control code such as {red} enclosed in quotes
Operator	An arithmetic operator to modify a term or join two terms

C64List recognizes the following operators in an expression (where a higher precedence number means higher precedence):

Operator	P
Unary +	9
Unary -	9
)	8
~ (unary)	7
> (unary)	6
< (unary)	6
*	5
/	5
+	4
-	4
&	3
|	2
(	1

C64List recognizes some special symbol names as well. When used as a symbol, the terms resolve as follows:

Symbol	Description
*	The current address that is being assembled (“PC”)
?	A 4-bit random value
??	An 8-bit random value
???	A 12-bit random value

????	A 16-bit random value

When used in an expression, local labels must be prefixed with a search direction and the @ symbol

Prefix	Description
<@	Identifies a local label and indicates a backward search direction
>@	Identifies a local label and indicates a forward search direction

The low- and high- byte unary operators can be used in combination with local label names as well. For example, the following instruction loads the low byte of the value of the local label @address in a backward search direction.

```
        lda #<<@address
```

Expressions may also contain C64 character codes, and C64List-style codes. For example, both of the following are valid:

```
        lda #'x'
        ldy #'{red}'
```

Remember that C64List must determine the addressing mode of an instruction when evaluating its parameter. Part of the addressing mode determination involves parentheses. The address mode is determined before the operator is evaluated. So be careful when using parentheses in the operand of an instruction to avoid confusing C64List. For example, this line of code is confusing:

```
        lda #(Start+10),y
```

C64List will interpret it as a zero page indirect indexed instruction, and probably give an error since Start is likely not a zero page address. But perhaps you actually wanted to do an absolute load from start+10 offset by the .y register. If that is what you meant, you will need to either remove the parentheses or compute the value into another symbol before using it as an operand:

```
        absoluteAddr = (Start+10)
        lda #absoluteAddr,y
```

In this simple case, the parentheses are not actually needed, but do keep this in mind when using complex expressions that require parentheses in combination with indexing.

Accessing symbols from BASIC

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 named FunctionName, use this format:

```
sys {sym:FunctionName}
```

To read a byte from memory address ByteName, use this format:

```
peek({sym:ByteName})
```

Here is a very brief example calling an assembly function from BASIC in C64List:

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

If you have a sharp eye, you may notice that the resulting .prg file often has what appears to be extra spaces between the sys keyword and the address. This is because the {sym:} directive converts any symbol into a 5-digit decimal value. Recall that C64List processes all BASIC before any assembly language. This means that during processing of the {sym:} directive C64List has no clue what the actual value of the symbol will be. It may be 1 digit or 5 digits long (the maximum value a symbol can take is 65535, which is 5 digits). So C64List reserves exactly 5 characters to account for whatever value the symbol ends up taking. For example, if **Screen** ends up being equal to 1024 decimal, then **POKE{sym:Screen},0** (with no spaces) will end up with two spaces between the POKE and the address: **POKE 1024,0**.

Code lines

The syntax of a code line contains anywhere from one to three elements: a **Label**, an **Instruction**, and an **Operand** (comments will have already been removed, so they don't count here). A code line may take any of the following forms:

```
LabelOnly:                      ;only a label is defined
LabelInstr:         clc         ;a label is defined; code is given
LabelInstrOper:     lda #$01    ;a label is defined; code + operand is given
                    clc         ;no label defined, but code is given
                    lda #$00    ;no label defined, but code + operand is given
```

The operand of the instruction is treated as an expression, and may consist of numeric or character constants, other symbols and operators.

Pseudo ops

Pseudo ops are "instructions" that may look much like 6510 instructions, but aren't. They are instructions to the assembler rather than the CPU. The assembler acts upon pseudo ops to produce

instructions or data, or modify the assembled code in some way. C64List recognizes a number of pseudo ops, each of which is discussed below.

---

orig <addr>

The **orig** pseudo op tells the assembler where in memory to place the next byte of code. The origin may only be moved forward in memory space. Any gap left by moving the origin is undefined, and typically filled with zeros. If one attempts to move the origin back in memory space, the assembler will complain and the code will not compile.

- The orig pseudo op is typically not advised in C64List mode since all machine code is automatically located immediately after the compiled BASIC code.
- An orig should always be explicitly set in casm mode; it defaults to 0x100 (the first non-zero page address, which is on the stack) if it not otherwise specified.

---

addrcheck <addr>

The **addrcheck** pseudo op inserts an address checkpoint in the code to prevent C64List from assembling code beyond the specified address. While linking, if the current address is greater than the address specified in the addrcheck's operand, an error will be generated. Multiple addrcheck pseudo ops may be inserted at different places in the code, with different addresses.

Use this pseudo op when there is a limit to the amount of code that can be generated before reaching a critical address. This feature will prevent C64List from assembling into the critical range.

You may also use this to check BASIC memory usage by inserting {asm} addrcheck <addr> {endasm} into your code. However, recall that all assembly is processed after BASIC, so no matter where you insert this into your BASIC code, it will apply to the *end* of your BASIC code.

---

align <addr>[, <fill>]

The **align** pseudo op tells the assembler to assemble the next instruction or data to an address such that it aligns with the given value, and fills the unused space with the optionally-given fill value. For example, if you embed sprite data in your code, you will need to align it to a 64-byte boundary. In this case, you would use the pseudo op **align 64**. You might also wish to fill the unused space with some value, such as $a5. If you don't specify anything for <fill>, the bytes are left undefined, and will typically be set to zero. You may also specify ?? to use a random fill value. Example of use:

```
      orig $1ff2
      rts
      align 64, $a5
      byte 0,0,0,255,255,255,0,0,0,255,255,255
```

The rts assembles to the address 1ff2, so the assembler fill enough bytes with the value $a5 so that the first 0 byte is assembled at $2000 like this:

```
1ff2: 60 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 00 00|`...............
2002: 00 ff ff ff 00 00 00 ff ff ff                  |..........
```

Although you may align on any value, powers of two are most useful. For example, 2 will align on an even address, and 0x100 will align on a page boundary. Aligning on 0 is meaningless, and doing so will cause no alignment to take place.

---

area <size>[, <fill>]

The **area** pseudo op tells the assembler to leave the specified number of bytes unused, and alternately, fill the unused bytes with a fill value. If you don't specify anything for <fill>, the bytes are left undefined, and will typically be set to zero. This example will create a 40-byte unused space filled with the value 32 and label the space SpacesSpace.

```
LastInstruction:    rts
SpacesSpace:        area 40, '{space}'
AFunction:          ldx #$00
                    lda SpacesSpace,x
```

---

byte <value>[, <value>[,…]]

The **byte** pseudo op inserts one or more bytes into memory at the current assembling address. Values can be any 8-bit numeric constants, symbols, special symbols, characters, or an expression consisting of any of the above. Examples:

```
orig $7000
FirstByteOfCode:          byte <(EndOfFile-FirstByteOfCode)
                          byte 1,2,3,4, '1', '2', '3', '4'
EndOfFile:
```

In the resulting memory below, you can see the number of bytes in the file (9), four numeric values (1 2 3 4), followed by the four PETSCII character values ('1' '2' '3' '4').

```
7000: 09 01 02 03 04 31 32 33 34                     |.....1234
```

---

word <value>[, <value>[,…]]

The **word** pseudo op inserts one or more word values into memory at the current assembling address. Values can be any 8- or 16-bit numeric constants, symbols, special symbols, character, or an expression consisting of any of the above. 16 bits are inserted into memory for each value. Examples:

```
                    orig $2800
FirstByteOfCode:    word 1, 2, '{yellow}', 'X'
HereWeAre = *
                    word ?, ??, ???, ????, EndOfFile, HereWeAre, <$1234
EndOfFile:
```

Here we can see: the 16-bit values (0001, 0002), 16-bit representations of the cursor control code for {yellow} ($009e) and 'X' ($0058), four random values ($0002, $001b, $09a3, $8b1f), the address EndOfFile ($2816), the address HereWeAre ($2808), and finally a word value containing the low byte of the constant $1234 ($0034).

```
2800: 01 00 02 00 9e 00 58 00 02 00 1b 00 a3 09 1f 8b|......X.....(...
2810: 16 28 08 28 34 00                              |.(.(4.
```

---

data <value>[, <value>[,…]]

The **data** pseudo op is similar in style and function to the byte and word pseudo ops. While **byte** *always* inserts 8-bit values, and **word** *always* inserts 16-bit values, **data** will insert *either* 8- *or* 16-bit values, depending on the sizes of the operands. This means that a combination of bytes and words may be added in a single line of code, rather than having to add multiple lines of byte and data pseudo ops. While this is a powerful feature, the user should be very careful to ensure that each operand is properly sized. Operand sizing is explained in *Operands*.

Values can be any 8- or 16-bit numeric constants, symbols, special symbols, character, or an expression consisting of any of the above. Examples:

```
ZeroPageAddr = $22
StackPageAddr = $01ff
SpadeChar = $61
ScreenLocation = $0428
       data $0001, $02, ZeroPageAddr, StackPageAddr, SpadeChar, ScreenLocation
```

Inspecting the output, we can see the following values:

- 16-bit: $0001
- 8-bit: $02
- 8-bit: $22
- 16-bit: $01ff
- 8-bit: 61
- 16-bit: $0428

```
0100: 01 00 02 22 ff 01 61 28 04                     |..."..a(.
```

---

ascii "<string>"[, "<string>"[,…]]

The **ascii** pseudo op inserts a string of characters into the code. Character constants and byte values may also be inserted. Multiple operands are accepted, separated by commas. The special C64List screen codes (such as {clear}{down:4}) are supported, and all characters are subject to the {alpha} mode that is currently in effect. Here's an example that shows a number of acceptable formats as well as using different alpha modes on the strings.

```
{alpha:Lazy}
String1:    ascii "Hello World!", 'x', "{0:3}"
```

```
{alpha:Alt}
String2:    ascii "Hello World!{down:3}", 0
{alpha:Ascii}
String3:    ascii "Hello World!{0}", 'X', 'y', 0
{alpha:Upper}
String4:    ascii "Goodbye!",0
```

In the hex dump we can see the four various strings inserted into memory in the four alpha modes that were specified.

```
0100: 48 45 4c 4c 4f 20 57 4f 52 4c 44 21 58 00 00 00|HELLO WORLD!X...
0110: c8 45 4c 4c 4f 20 d7 4f 52 4c 44 21 11 11 11 00|.ELLO .ORLD!....
0120: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 00 58 79 00|Hello World!.Xy.
0130: c7 cf cf c4 c2 d9 c5 21 00                     |.......!.
```

---

hex <value>[<value>[…]]

bin <value>[<value>[…]]

The **hex** and **bin** pseudo ops insert bytes into the code. While they are similar to the **byte** pseudo op, they are specifically geared at putting hexadecimal or binary notated values into memory. Therefore, only valid hexadecimal or binary values (without any preceding sign) are accepted, and must be separated by spaces. Here's an example of defining part of a sprite:

```
            orig $1ffe
            align $40, $ff
Sprite:     hex 01 ff 80
            bin 00000001 11111111 10000000
```

In the memory dump we can see that the hex and binary values were inserted into memory:

```
1ffe: ff ff 01 ff 80 01 ff 80                        |........
```

---

bits <bit characters>

The **bits** pseudo op is very similar to the **bin** pseudo op in that it inserts binary values into the code. However, this feature was designed to insert character, sprite, and other bitmap data directly into your source code. Instead of taking '0's and '1's as parameters, however, the **bits** pseudo op takes characters that better visualize the graphical nature of the data.

Monochromatic bitmaps are represented by a set of two characters ("." and "*"). This allows viewing, creating, and editing graphical data right in the source code, without requiring any special sprite or character editor software.

Multicolor bitmaps are likewise represented by a set of four characters: ('-', '/', '\', and '|'). These characters provide a visual shading that represent the four bit patterns available to multicolor graphics on the C64.

Unlike **bin** and **hex** pseudo ops where whitespace is required between bytes, the **bits** pseudo op does

not use whitespace to delineate byte groupings. Therefore, whitespace can be used as desired anywhere in the data, and C64List will ignore (remove) any such whitespace when compiling the data into `.prg` format. Instead, it groups each set of 4 or 8 consecutive graphical characters into bytes. C64List requires that the number of characters in a **bits** pseudo op corresponds to a multiple of 8 bits in the output data, and will report an error if there are missing or extra characters.

For monochromatic bitmaps

Character	Converts to
.	0 (a single 0 bit)
*	1 (a single 1 bit)

For multicolor bitmaps

Character	Converts to
-	00 (2 bits)
/	01 (2 bits) mnemonic: think of '/' leaning to the right toward the 1 in 01
\	10 (2 bits) mnemonic: think of '\' leaning to the left toward the 1 in 10
|	11 (2 bits) mnemonic: think of '|' as balanced between the two 1s in 11

To define a 24x21 monochrome sprite, simply write 21 rows of 24 "*" or "." characters. To define a 12x21 2-color sprite, write 21 rows of 12 "-", "/", "\", or "|" characters. In either case, don't forget to add the 64th unused byte before defining another sprite.

To define an 8x8 monochrome character, simply write 8 rows of 8 the monochrome characters. To define a 4x8 multicolor character, write 8 rows of 4 multicolor characters.

Examples of use:

```
orig $2000

;monochromatic sprite
bits ************************
bits **......................**
bits **......................**
bits **.....**......**.....**
bits **.....**......**.....**
bits **......................**
bits **......................**
bits **.........**.........**
bits **.........**.........**
bits **......................**
bits **......................**
bits **.....**......**.....**
bits **.....**......**.....**
bits **......................**
bits **......................**
bits **......................**
bits **......................**
bits **......................**
bits **......................**
bits ************************
byte 0       ;byte #64

;4-color sprite
```

```
bits ////////////
bits ------------
bits \\\\\\\\\\\\
bits ||||||||||||
bits ||||||||||||
bits \\\\\\\\\\\\
bits ------------
bits ////////////
bits -/\|-/\|-/\|
bits -/\|-/\|-/\|
bits -/\|-/\|-/\|
bits ////////////
bits ////////////
bits /||||||||||/
bits /||||||||||/
bits /\\\\\\\\\\/
bits /\\\\\\\\\\/
bits /----------/
bits /----------/
bits ////////////
bits ////////////
byte 0       ;byte #64

;8x8 monochromatic character definition
bits ********
bits **....**
bits **....**
bits **....**
bits **....**
bits **....**
bits **....**
bits ********
```

---

embed <filename>[, offset[, length]]

The **embed** pseudo op inserts the contents of the specified file into the current program. Filename can specify a windows file system file, or it can specify a file within a .d64 file using the d64::C64 filename flexible syntax. C64List uses the search path to locate the file, so you can use relative file paths.

If offset and length parameters are not given, the entire file is inserted. If an offset is given, the specified number of bytes will be discarded before beginning to insert data. This could be useful to drop the load address from a .prg file before adding it, or you could index into the file to get some resource such as sprite data. The length parameter, if used, tells C64List how many bytes you want to insert from the file.

Part 4: Reference

The following sections list the various features of C64List in tabular form for easy reference.

Appendix: Preprocessor Directives

These directives are completely processed and resolved while C64List is loading the source files before the loaded source is passed to the BASIC tokenizer or assembler. Each of these directives is replaced by a string of some kind. For example the directive {include:file.bas} is replaced by the entire contents of file.bas. {def:var} is replaced by an empty string, and {usedef:var} is replaced by the value assigned to the parser variable var. {$:10} is replaced by 16.

Source Loader Directives may be placed anywhere as needed in the source code.

Directive	Effect	Replaced by
{include:<*file*>}	When loading text-formatted files, includes another text-formatted file <*file*>	The contents of the specified file
{uses:<*file*>}	Similar to {include:}, but automatically checks to see if <*file*> has already been included. If so, does not include it again.	The contents of the specified file
{addsearchpath:<*path*>}	Adds <*path*> to the {include} and {uses} search path	<empty>
{buildrev:<*path*>}	Loads the build rev ID from <*path*> into the parser variable **__BuildRev**, and causes the value in the file to be incremented after a successful build with no errors.	<empty>
{def:<*var*>[=<*val*>]}	Defines the parser variable <*var*>, and optionally sets it to the string <*val*>	<empty>
{undef:<*var*>}	Undefines the parser variable <*var*>	<empty>
{ifdef:<*var*>}	Begins a block of lines which are only processed if <*variable name*> is currently **defined** (see {def})	<empty>
{ifdnef:<*var*>}	Begins a block of lines which are only processed if <*var*> is currently **not defined**.	<empty>

{else}	Ends an {ifdef} or {ifndef} block and immediately starts another block with the opposite condition	<empty>
{endif}	Ends an {ifdef}, {ifndef}, or {else} block and returns to normal processing.	<empty>
{usedef:<*var*>}	Insert the value of the specified parser variable	The current value of the specified parser variable
{asm}	Route the following source code to the assembler instead of the BASIC tokenizer, until the an {endasm} directive is found. {asm} and {endasm} are ignored in .asm files.	<empty>
{endasm}	Stop routing the source code to the assembler, and route it to the BASIC tokenizer instead. {asm} and {endasm} are ignored in .asm files.	<empty>
{macro:<*name*> <*params*>}	Start a macro assembly definition	<empty>
{endmacro}	End a macro assembly definition	<empty>
{$:<*hexvalue*>}	Replaces the directive in the code with the decimal equivalent of <*hexvalue*>. Note that this directive is syntactically and operationally different from the BASIC numeric token directives {$<*hexvalue*>}, {%<binaryvalue>}, and {<decimalvalue>}.	The decimal equivalent of the specified hexadecimal value

{%:<*binaryvalue*>}	Replaces the directive in the code with the decimal equivalent of <*binaryvalue*>. Note that this directive is syntactically and operationally different from the BASIC numeric token directives {$<*hexvalue*>}, {%<binaryvalue>}, and {<decimalvalue>}.	The decimal equivalent of the specified binary value
{info:<*message*>}	Output <*message*> asn in informational note when the directive is processed. The message will only be visible if -verbose is in effect	<empty>
{warning:<*message*>}	Output a warning <*message*> when the directive is processed. Signaled as an actual warning in C64List.	<empty>
{error:<*message*>}	Output an error <*message*> when the directive is processed. Signaled as an actual error in C64List, and stops processing.	<empty>
{fatal:<*message*>}	Output a fatal error <*message*> when the directive is processed. Signaled as an actual error in C64List, and stops processing. Currently same behavior as {error:}	<empty>
{‘directive}	Any directive can be ignored by placing a tick immediately following the open curly brace	<empty>

*Note as of C64List version 4.00: although macros may be defined, C64List is not yet able to use them.

Appendix: String Engine Directives

String Engine directives are valid in BASIC and assembly source code, but only inside quoted strings. They describe characters inside C64-style strings--such as cursor codes--in a way that is easily read by humans. Many String Engine Directives have alternate spellings to make them as easy as possible to remember. All of the listed cursor control codes may be suffixed with a colon followed by a repetition value (such as

“{down:10}”)

Directive	Description	C64 Keystroke	Glyph
{f1}	F1	F1	□
{f2}	F2	Shift-F1	□
{f3}	F3	F3	□
{f4}	F4	Shift-F3	□
{f5}	F5	F5	□
{f6}	F6	Shift-F5	□
{f7}	F7	F7	□
{f8}	F8	Shift-F7	□
{stop}	Insert mode character for RUN/STOP key	RUN/STOP	Ã
{runstop}			
{shft ret}	Shifted return character	Shift-RETURN	
{shift return}			
{switchdisable}	Character set lock	CTRL-H	
{switchdis}			
{switchenable}	Character set unlock	CTRL-I	
{switchena}			
{lowercase}	Switch to lowercase character set		
{lower}			
{uppercase}	Switch to uppercase character set		
{upper}			
{left}	Move cursor left	Shift-CRSR rt	□
{lt}			
{right}	Move cursor right	CRSR rt	Ý
{rt}			
{up}	Move cursor up	Shift-CRSR dn	□
{down}	Move cursor down	CRSR dn	□
{dn}			
{home}	Move cursor home	CLR/HOME	Ó
{clear}	Clear screen and move cursor home	Shift-CLR/HOME	□
{clr}			
{insert}	Insert mode character (printable insert key)	Shift-INST/DEL	□
{inst}			
{backspace}	Insert mode character	INST/DEL	Ô

	(printable delete key)		
{bksp}			
{delete}			
{del}			
{rvrs on}	Invert the printing colors	CTRL-9	◘
{rvs on}			
{reverse on}			
{rvrs off}	Restore normal printing colors	CTRL-0	◘
{rvs off}			
{reverse off}			
{black}	Set cursor color to black	CTRL-1	◘
{white}	Set cursor color to white	CTRL-2	Å
{red}	Set cursor color to red	CTRL-3	Ü
{cyan}	Set cursor color to cyan	CTRL-4	◘
{purple}	Set cursor color to purple	CTRL-5	◘
{green}	Set cursor color to green	CTRL-6	Þ
{blue}	Set cursor color to blue	CTRL-7	ß
{yellow}	Set cursor color to yellow	CTRL-8	◘
{orange}	Set cursor color to orange	C=-1	◘
{brown}	Set cursor color to brown	C=-2	◘
{lt. red}	Set cursor color to light red	C=-3	◘
{lt red}			
{ltred}			
{light red}			
{gray1}	Set cursor color to dark gray	C=-4	◘
{gray 1}			
{dk. gray}			
{dk gray}			
{dkgray}			
{dark gray}			
{grey1}			

{grey 1}			
{dk. grey}			
{dk grey}			
{dkgrey}			
{dark grey}			
{gray2}	Set cursor color to medium gray	C=-5	◘
{gray 2}			
{gray}			
{grey2}			
{grey 2}			
{grey}			
{lt. green}	Set cursor color to light green	C=-6	◘
{lt. blue}	Set cursor color to light blue	C=-7	◘
{gray3}	Set cursor color to light gray	C=-8	◘
{gray 3}			
{lt. gray}			
{lt gray}			
{ltgray}			
{light gray}			
{grey3}			
{grey 3}			
{lt. grey}			
{lt grey}			
{ltgrey}			
{light grey}			
{^}	The up arrow character	Up arrow	^
{up arrow}			
{back arrow}	The back arrow character	Back arrow	_
{left arrow}			
{pound}	The British Pound symbol	British pound	\
{british pound}			
{shft pound}	The character you get when shifting the pound key	Shift-pound	¨
{shift pound}			

{ctrl pound}	The character you get when hitting ctrl+pound key	Ctrl-pound	©
{cmdr pound}	The character you get when hitting cmdr+pound key	C=-pound	©
{shft space}	Shifted space character	Shift-Spacebar	
{shift space}			
{space}	Spacebar	Spacebar	
{quote}	Double quote character	Quote	"
{return}	The return character {13}	Return	
{null}	A null character {0}	Character 0	
{pi}	The PI symbol	PI	~
{lift}	Non-printing modifier virtual key only		
{shft}	Non-printing modifier shift key only		
{shift}			
{cmdr}	Non-printing modifier commodore key only		
{commodore}			
{ctrl}	Non-printing modifier control key only		
{control}			
{<*byte*>}	Insert specified <*byte*> value directly into string		

Notes:

- Shaded entries indicate acceptable alternate spellings for main entry above
- These characters have only alternates (text-to-prg only)
    - Space
    - Quote
    - Return
    - Null (character 0)
    - PI
- Non-printing Modifier keys
    - Lift
    - Shift
    - Commodore
    - Control
- {<*byte*>} may be prefixed with a dollar sign ("$") or percent sign ("%") for hexadecimal or binary notation.
    - This directive is very similar to the {<*token*>} BASIC directive.
    - This directive is syntactically and functionally different from the Preprocessor directives

{$:<*value*>} and {%:<*value*>}

Appendix: BASIC Tokenizer Directives

BASIC tokenizer directives are only valid in BASIC code, and control various aspects of how the BASIC code is interpreted, tokenized, and numbered.

Directive	**Description**
{:<*label*>}	A label starts with a colon. It marks a line where GOTOs and GOSUBs may be called.
{:<*numeric label*>}	Same as a regular label, except that the numeric value is used as a line numbering hint when C64List is auto-numbering the code.
{renumber}	Request that the BASIC program be renumbered.
{number:<*line number*>}	Causes the next line of code to have the specified line number. Restrictions: • 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*>}	Causes the line numbering to increment by the specified value. Activates on the second line of code after the directive. Restrictions: • 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 was 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.
{crunch:<*switch*>}	Turns on or off crunch mode.
{autospace:<*switch*>}	Turns on or off autospace mode: strategically insert spaces in the BASIC code to make it more readable. Has the opposite effect of crunch mode.
{crsrcodes:<*switch*>}	Turns on or off cursor code conversion when converting to text. Useful in a preface file.

Directive	Description
{crsrcount:<*switch*>}	Turns on or off cursor code collation counts when converting to text. Useful in a preface file.
{remremoval:<*switch*>}	Turns on or off the REMark removal in-line in your BASIC code.
{loadaddr:<*address*>}	Begin compiling BASIC (or ML) at <address> rather than $0801. Restriction: must be placed before any code in the file.
{tokenizer:<tkn>=<kywrd>	Adds custom BASIC tokens for BASIC extensions.
{quoter:<*tkn*>=<*kywrd*>	Adds custom screen tokens for characters in quote mode
{sym:<*symbol*>}	Refer to a symbol in the assembly symbol table
{alpha:<*mode*>}	Sets the alpha character handling mode, which tells C64List how to handle printable characters that are inside quotes. This directive itself does not go inside quotes. Valid settings for <*mode*> are: • Ascii (no character conversions will be made) • Lazy (for programming in the standard character set) • Alt (for programming in the alternate character set) • Poke (convert characters to screen POKE codes) • PokeAlt (Convert character to alternate set POKE codes) • Upper (convert lowercase characters to uppercase) • Lower (convert upper characters to lowercase) • ~~Invert~~ (invert upper and lowercase characters) Note: The default value is Lazy Note: Invert has been deprecated in favor of Alt
{var:<*supervar*>[=<*vv*>]}	Name or use a supervariable
{assign:<*label*>=<*number*>}	Assign an undefined label to an uncoded line number. This is useful if line number <*number*> exists somewhere outside the codebase. Doesn't reserve the line number per-se, but it allows GOTO <*label*> and GOSUB <*label*> without generating errors during tokenization.
<keycase:<*case*>}	When converting to text, render keywords in upper or lower case. Valid options for <case> are: • l|lo|lower = render in lowercase • u|up|upper = render in uppercase Note: uppercase is default
<varcase:<*case*>}	When converting to text, render variable names in upper or lower case. Valid options for <case> are: • l|lo|lower = render in lowercase • u|up|upper = render in uppercase Note: uppercase is default
{shortkeywords:<*switch*>}	Not implemented yet
{<*token*>}	Inserts the byte value of <*token*> directly into the BASIC code. <*token*> may be prefixed by a dollar sign ("$") or percent sign ("%") to use hexadecimal or binary notation.

Directive	Description
	Notes: • This directive is very similar to the {<*token*>} String Engine directive. • This directive is syntactically and functionally different from the Preprocessor directives {$:<*value*>} and {%:<*value*>}
{pi}	The {pi} directive is actually a BASIC keyword. Since the pi symbol is a C64 BASIC keyword, but the Windows keyboard does not have a pi symbol, {pi} allows you to insert the keyword into your source code. When converting a .prg file back to text format, the pi keyword will be re-converted to the {pi} directive.

Appendix: Assembler Directives

Assembler Directives: These directives are only valid in assembly source code (in .asm files or inside {asm}/{endasm} directive pairs).

Directive	Description
{alpha:<*mode*>}	Same as the BASIC Tokenizer Directive of the same name
{symdemote:<*switch*>}	When enabled, demote all newly-defined Important symbols to General. Useful when including an existing assembly file that contains important-level symbols, but you aren't interested in seeing them in a symbol table.

Appendix: C64List command line parameters

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.
-bin[:filename[.ext]]	Same as -prg, except omits the load address from the file.
-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. -hex:con outputs to the console instead of a file.
-lst[:filename[.ext]]	Disassemble and output the loaded machine language program in text format. The default name will be <input filename>.lst if not specified

Command Line Option	Description
	here. -lst:con outputs to the console instead of a file.
-h	Output a help screen and exit immediately
-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. -txt:con outputs to the console instead of a file.
-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. -lbl:con outputs to the console instead of a file.
-sym[:filename[.ext]]	Create a symbol file from all the assembly symbols that were defined in the source code. The default name will be <input filename>.sym if not specified here. -sym:con outputs to the console instead of a file.
-d64:D64Name::PRGNAME	Save the program into a file named “PRGNAME” inside a .D64 file named D64Name.prg.
-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
-labels	Outputs a list of the labels that were defined in the source.
-ovr	Overwrite an existing file if necessary
-f	Same as –ovr
-loadext:<ext>	If the input file’s extension is not recognized by C64List (or is missing), the -loadext command line option can specify what type of file it is. This

Command Line Option	Description
	allows loading such files without renaming them first. -loadext is ignored if you are loading from a file inside a .d64, since only .prg-type files can be loaded from .d64.
-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. May be set to one of Off|On|List. Specifying `-verbose` with no parameter is the same as `-verbose:on`. Not specifying –verbose at all has the same effect as `-verbose:off`. `-verbose:list`, which outputs each line of text while processing it.
-alpha:<mode>	Sets the alpha mode on the command line. May be set to one of: Off|Normal|Alt|Upper|Lower. Please see the section entitled *Mixed character case support* for details.
-range:<start>[-<end>]	When outputting `.hex` or `.asm` files, this allows you to limit the memory range which is output to the file.
-pref:<filename>	Specifies a (text-formatted) preface file containing such things as tokenizer directives. This function is useful for loading tokenized .prg files that contain customized tokens so the custom tokens are rendered as the proper custom keywords.
-symlvl:<level>	Filter .sym symbol file output to the desired level. Options are !|-|@|auto. • **!** or **important** or **high** = only output Important labels • **-** or **general** or **normal** = output general and important labels • **@** or **local** or **low** = output all symbol levels
-sym3	Output the symbol file surrounded with {asm} and {endasm} so that C64List 3.0 would be able to use the file as a source file.
-def:<name>	Define the parser variable named <name> on the command line. This does the same thing as the {def:<name>} directive. A command line may contain multiple -def: parameters.
-supervariables	When converting an existing `.prg` file to a `.txt` or `.lbl`, identifies all BASIC variables and converts them to Supervariables.

Command Line Option	Description
-labels	Dump the BASIC label table.