



# WP 34s Assembler Tool Suite User Guide

This file is part of **WP 34S**.

**WP 34S** is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

**WP 34S** is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with **WP 34S**. If not, see <http://www.gnu.org/licenses/>.

## Table of Contents

|       |  |    |
|-------|--|----|
| 1     | INTRODUCTION.....  | 4  |
| 1.1   | GOALS.....   | 4  |
| 1.2   | NOTES.....   | 4  |
| 1.3   | ACKNOWLEDGEMENTS.....                                      | 5  |
| 2     | LIST OF ACRONYMS AND DEFINITIONS.....                      | 6  |
| 3     | OVERVIEW.....  | 7  |
| 3.1   | SUPPORTED FEATURE SET.....                                 | 7  |
| 4     | ASSEMBLER / DISASSEMBLER.....                              | 8  |
| 4.1   | DISASSEMBLY INTRODUCTION.....                              | 8  |
| 4.2   | ASSEMBLY INTRODUCTION.....                                 | 9  |
| 4.3   | DISASSEMBLER INSTRUCTIONS.....                             | 9  |
| 4.4   | ASSEMBLER INSTRUCTIONS.....                                | 10 |
| 5     | SYMBOLIC PREPROCESSOR.....                                 | 13 |
| 5.1   | RUNNING THE PREPROCESSOR STAND-ALONE.....                  | 13 |
| 5.1.1 | STAND-ALONE PREPROCESSOR.....                              | 13 |
| 5.1.2 | RUNNING PREPROCESSOR FROM WITHIN ASSEMBLER .....           | 13 |
| 5.2   | JMP PSEUDO INSTRUCTION .....                               | 14 |
| 5.2.1 | JMP TARGET OFFSET GREATER THAN MAXIMUM OFFSET ALLOWED..... | 17 |
| 5.3   | GSB PSEUDO INSTRUCTION .....                               | 17 |
| 5.4   | LBL-LESS TARGETS.....                                      | 18 |
| 5.5   | DOUBLE QUOTED TEXT STRINGS.....                            | 19 |
| 6     | LIBRARY MANAGER.....                                       | 21 |
| 6.1   | LIBRARY MANAGER OPTIONS.....                               | 21 |
| 6.2   | LIBRARY MANAGER EXAMPLES.....                              | 21 |
| 6.2.1 | GENERATING A CATALOGUE.....                                | 22 |
| 6.2.2 | DELETING ONE OR MORE PROGRAMS.....                         | 22 |
| 6.2.3 | ADDING OR REPLACING ONE OR MORE PROGRAMS.....              | 23 |
| 6.2.4 | CONVERTING BINARY LIBRARIES TO FLASH FORMATS.....          | 24 |
| 6.2.5 | DEALING WITH STATE FILE LIBRARIES.....                     | 25 |
| 7     | ADDITIONAL INFORMATION.....                                | 27 |
| 7.1   | GENERATING REFERENCE OP-CODE LIST .....                    | 27 |
| 7.2   | MIGRATING PROGRAMS BETWEEN OP-CODE REVISIONS.....          | 27 |
| 7.3   | AUTOMATICALLY LOCATING OP-CODE MAP.....                    | 28 |
| 8     | EXAMPLES LIBRARY.....                                      | 29 |
| 9     | SOURCE OF PERL PACKAGES.....                               | 30 |
| 9.1   | LINUX.....   | 30 |
| 9.2   | WINDOWS.....   | 30 |
| 9.2.1 | CYWIN.....   | 30 |
| 9.2.2 | STRAWBERRY PERL.....                                       | 30 |
| 9.2.3 | ACTIVE STATE PERL.....                                     | 30 |
| 9.2.4 | NATIVE WINDOWS EXECUTABLE.....                             | 31 |
| 9.3   | MAC O/S.....   | 31 |
| 10    | HELP.....  | 32 |

## Table of Figures

|             |  |   |
|-------------|--|---|
| FIGURE 4.1: | DISASSEMBLER EXAMPLE COMMAND-LINE..... | 8 |
|-------------|--|---|

|  |    |
|--|----|
| FIGURE 4.2: DISASSEMBLED SOURCE EXAMPLE.....                               | 8  |
| FIGURE 4.3: ASSEMBLER EXAMPLE COMMAND-LINE.....                            | 9  |
| FIGURE 4.4: GENERIC DISASSEMBLER COMMAND-LINE.....                         | 9  |
| FIGURE 4.5: DISASSEMBLER COMMAND-LINE FOR 2 LABEL ASTERISKS.....           | 9  |
| FIGURE 4.6: SOURCE EXAMPLE WITH LABEL ASTERISKS.....                       | 10 |
| FIGURE 4.7: DISASSEMBLER COMMAND-LINE WITH STEP NUMBERS SUPPRESSED.....    | 10 |
| FIGURE 4.8: SOURCE EXAMPLE WITH NO LINE NUMBERS.....                       | 10 |
| FIGURE 4.9: GENERIC ASSEMBLER COMMAND-LINE.....                            | 11 |
| FIGURE 4.10: ASSEMBLER SOURCE WITH COMMENT STYLES.....                     | 11 |
| FIGURE 4.11: ASSEMBLING MULTIPLE SOURCE FILES TO ONE IMAGE.....            | 12 |
| FIGURE 5.1: RUNNING WP 34S PP SOURCE THROUGH WP 34S PP.....                | 13 |
| FIGURE 5.2: RUNNING WP 34S PP THROUGH THE WP 34S ASSEMBLER .....           | 14 |
| FIGURE 5.3: ORIGINAL 8-QUEENS BENCHMARK CODE.....                          | 14 |
| FIGURE 5.4: JMP PSEUDO INSTRUCTION EXAMPLE – WP 34S PP SOURCE .....        | 15 |
| FIGURE 5.5: JMP PSEUDO INSTRUCTION EXAMPLE – WP 34S PP OUTPUT.....         | 16 |
| FIGURE 5.6: EFFECT OF BACK/SKIP BEYOND MAXIMUM ALLOWABLE OFFSET.....       | 17 |
| FIGURE 5.7: LBL-LESS EXAMPLE – WP 34S PP SOURCE.....                       | 18 |
| FIGURE 5.8: LBL-LESS EXAMPLE – WP 34S PP OUTPUT.....                       | 19 |
| FIGURE 5.9: ORIGINAL ALPHA STRINGS PROGRAMS.....                           | 19 |
| FIGURE 5.10: DOUBLE QUOTED ALPHA STRING EXAMPLE.....                       | 20 |
| FIGURE 5.11: DOUBLED QUOTED ALPHA STRING DISASSEMBLY.....                  | 20 |
| FIGURE 6.1: GENERATING A CATALOGUE FROM A BINARY LIBRARY.....              | 22 |
| FIGURE 6.2: DELETING PROGRAMS FROM BINARY LIBRARY.....                     | 23 |
| FIGURE 6.3: DELETING PROGRAMS FROM BINARY LIBRARY – ALTERNATE.....         | 23 |
| FIGURE 6.4: ADDING AND REPLACING PROGRAMS FROM BINARY LIBRARY.....         | 24 |
| FIGURE 6.5: ADDING PROGRAMS FROM BINARY LIBRARY.....                       | 24 |
| FIGURE 6.6: CONVERTING LIBRARY FROM STATE TO FLASH FORMAT.....             | 25 |
| FIGURE 6.7: OPERATING ON A STATE FILE.....                                 | 25 |
| FIGURE 6.8: CONVERTING STATE TO FLASH FORMAT.....                          | 26 |
| FIGURE 7.1: ABRIDGED VIEW OF OP-CODE SYNTAX TABLE.....                     | 27 |
| FIGURE 7.2: COMMAND-LINE TO DISASSEMBLE USING A SPECIFIED OP-CODE MAP..... | 28 |
| FIGURE 7.3: COMMAND-LINE TO ASSEMBLE USING A SPECIFIED OP-CODE MAP.....    | 28 |
| FIGURE 9.1: RUNNING SCRIPT UNDER STRAWBERRY PERL.....                      | 30 |
| FIGURE 9.2: RUNNING SCRIPT UNDER ACTIVE STATE PERL.....                    | 31 |
| FIGURE 9.3: RUNNING THE WINDOWS EXECUTABLE .....                           | 31 |
| FIGURE 10.1: INVOKING THE HELP SCREENS.....                                | 32 |

## List of Tables

## 1 INTRODUCTION

The **WP 34S** is a “repurposing” project designed for the **HP-20b Business Consultant** and **HP-30b Business Professional** calculators designed by Hewlett-Packard. These calculators are based on an Atmel ARM 7 single chip device containing 128KB of programmable flash memory.

HP made a serial port available under the back cover-plate of these calculators and through this serial port it is possible to “reflash” the calculator's application software using a software download tool made available by Atmel<sup>1</sup>.

The **WP 34S** project team designed new S/W for the calculator chassis to convert the **HP-20b** and/or **HP-30b** calculators into highly competent Reverse Polish Notation (RPN) programmable scientific calculators (see <http://sourceforge.net/projects/wp34s/> for more details). The **WP 34S** project has been done with the knowledge – and occasional assistance – of HP staff.

This Assembler Tool Suite is targeted at helping to design, archive, distribute, and maintain those **WP 34S** FOCAL<sup>2</sup> programs.

### 1.1 Goals

The **WP 34S** Assembler Tool Suite was designed to accomplish several goals:

1. Archiving and distribution of **WP 34S** FOCAL programs in a distributable source form (text files) as opposed to binary images.
2. At least in its early days, the **WP 34S** design was in flux and the binary op-code map was known to fluctuate from revision to revision. This meant that binary flash images might not be compatible between calculator F/W revisions. By allowing the FOCAL programs to be saved in an ASCII source format, they can be more easily ported between revisions.
3. Calculator programmers often develop a collection of FOCAL programs to accomplish many tasks. The assembler is able to process several source files and concatenate the result into a single flash image allowing some form of primitive library management.
4. A slightly more formal library manage tool was added to the suite to allow FOCAL programs to be inserted, deleted, or *refreshed* within a binary library.

As the **WP 34S** developed, a section of the internal programming (XROM) was redesigned to be in a form similar to the user's FOCAL programming language. This tool suite was also designed to support the XROM program source but it is not expected nor anticipated that users will ever access these features of the tools. Thus they will not be documented in this manual – and they will remain the domain of the **WP 34S** development team.

### 1.2 Notes

This manual may show examples from specific SVN revisions of tools and/or files. It is intended to be representative of the current tool. From time-to-time, the examples may be updated when the clarity of the information needs to be improved. Hence, the fact that the cover picture<sup>3</sup> shows SVN 2441 does not indicate that this is the last – or only – version supported.

<sup>1</sup> There is also an alternative, and most agree superior, tool available in at the Source Forge site: <http://sourceforge.net/projects/wp34s/files/FlashTool/>

<sup>2</sup> HP RPN key-stroke programs are sometimes known as 'FOCAL' programs, though it is my understanding that, strictly speaking, FOCAL specifically refers to **F**orty **O**ne **C**alculator **L**anguage. However, with forgiveness of the community, I will use the term FOCAL when discussing **WP 34S** user key-stroke programs as it is a more compact way of referring to them. Furthermore, this document will use the terms *assemble* and *disassemble* in connection with the FOCAL programs. *Assemble* refers to converting FOCAL programs to the binary token representation used internally by the calculator F/W. *Disassemble* refers to reversing that process and converting the binary token representation back into user readable FOCAL source.

<sup>3</sup> The cover image was shamelessly stolen from a screen shot of the **WP 34S** emulator. The author begs forgiveness for this and other sins. The author will not be updating the cover image with every new revision of the calculator so the version number shown here is **not** indicative of the version of this tool.

The Assembler Tool Suite is located in the “./trunk/tools/” directory within the SVN development tree. The examples in this document assume that the user has their PATH correctly set up to reach this directory. Since that is O/S (and sometimes command shell) specific, the user is directed elsewhere for instructions of setting this up in their installation.

### 1.3 Acknowledgements

Without the tireless efforts of Walter Bonin, Paul Dale, and Marcus von Cube (the “Gang of Three”<sup>4</sup>) the **WP 34S** would not exist. These guys seem motivated to work around the clock on this device and have answered my questions with a degree of patience that is impressive! Cyrille de Brebrisson, Tim Wessman and their cadre of co-conspirators at HP must be mentioned for their foresight in allowing the **HP-20b** and **HP-30b** calculators a route to flash reprogramming and for providing a wealth of material and information about the machines – including the original emulator and SDK.

Since the original version of these tools came about, several additional folks have joined the **WP 34S** team and they too deserve some acknowledgement. Fellow Canadian, Dominic Richens now maintains the V2 source code base and Pascal Méheut is working on the multi-platform emulator support.

---

<sup>4</sup> Thanks to Gerry Schultz for coming up with a better title than “techo-trinity”.

## 2 LIST OF ACRONYMS AND DEFINITIONS

|       |  |
|-------|--|
| ASM   | <b>WP 34S</b> Assembler/Disassembler (Perl script: wp34s_asm.pl)                             |
| CPAN  | Comprehensive Perl Archive Network ( <a href="http://www.cpan.org">http://www.cpan.org</a> ) |
| dat   | Filename extension of a binary program image from flash or RAM used by the emulator          |
| F/W   | Firmware   |
| FOCAL | <u>F</u> orty <u>O</u> ne <u>C</u> alculator <u>L</u> anguage                                |
| LIB   | <b>WP 34S</b> Library Manager (Perl script: wp34s_lib.pl)                                    |
| op    | Op-code definition file extension  |
| PP    | <b>WP 34S</b> Symbolic Preprocessor (Perl script: wp34s_pp.pl)                               |
| RPN   | Reverse Polish Notation  |
| SDK   | Software Development Kit   |
| SVN   | Subversion – a revision control tool   |
| S/W   | Software   |
| wp34s | Filename extension of a <b>WP 34S</b> source file (by convention)                            |

### 3 OVERVIEW

The **WP 34S** Assembler Tool Suite uses the op-code format as described in the wp34s.op file. This file is automatically generated by the main F/W build process. See Section 7.1 Generating Reference Op-code List for more details.

This suite consists of 3 tools:

**Assembler/Disassembler (ASM):** This dual-function tool performs both assembly and disassembly tasks – depending on the command line parameters used. It is currently a single pass, direct translation tool. In its assembler mode, it converts FOCAL program written in user readable text into the binary tokens used by the calculator internals. In its disassembler mode, it reverses the process and converts the binary tokens back into user readable FOCAL.

**Preprocessor (PP):** This tool is a front-end that converts some symbolic coding-styles into 'pure' user FOCAL code that can be translated by the Assembler. It also eases specifying of program branches, text strings, and several other features.

**Library Manager (LIB):** The LIB tool allows the user to manipulate binary library images. This tool allows the addition, replacement, and/or deletion of named programs to/from a library image. It also allows the user limited ability to convert between the two forms of binary images used by the emulator<sup>5</sup>. FOCAL programs in an emulator state file can be converted to flash image and, conversely, flash images can be inserted into an existing emulator state file.

The Assembler Tool Suite tool chain is written entirely in Perl (version 5.8+) in order to be portable across many platforms. Portions of the tool chain have been compiled into a Windows native EXE for those installations that cannot afford the 100-200 MB of disk space for a Perl interpreter package. See Section 7.3 for more details.

#### 3.1 Supported Feature Set

- Will assemble one or more ASCII source files into a **WP 34S** binary image.
- Will disassemble a **WP 34S** binary image into an ASCII FOCAL source listing.
- Can accept alternate **WP 34S** op-code maps to translate source and/or flash images between **WP 34S** SVN revisions.
- Can output a "op-code syntax guide" to help with writing accurate FOCAL source files.
- Will automatically locate the most current op-code map if updated via SVN.
- The symbolic preprocessor script can be used to simplify **WP 34S** source code.
- The library manager can be used to adjust the content of a binary library image.

---

<sup>5</sup> The emulator uses 2 forms of binary images. The first is a flash image and the second is the emulator state image. The second form contains the last state of the emulator when the tool was closed. This is the contents of the registers, the program RAM section, etc.. The LIB tool can convert a flash image to a emulator state provided it is given a "seed" emulator state file to modify. This seed file can be generated by simply opening and closing the emulator.

## 4 ASSEMBLER / DISASSEMBLER

The ASM tool is both an assembler and a disassembler combined into one script. For many people, the disassembler function may be the first mode they use – and this will likely be to disassemble a flash image recovered directly from the **WP 34S** calculator and/or the emulator.

### 4.1 Disassembly Introduction

The disassembler reads a binary image created with the “SAVE”, “PST0”, “P↔”, etc. op-codes, or via simply exiting the emulator<sup>6</sup>. These binary images can be generated directly using the emulator or can be downloaded from the calculator using the serial cable, SAM-BA, and the instructions in the main **WP 34S** documentation.

The disassembler is run from a command line shell. For example, to disassemble a set of programs saved from flash, the following command may be used:

```
$ wp34s_asm.pl -dis wp34s-lib.dat -o myProgs.wp34s
```

**Figure 4.1: Disassembler Example Command-Line**

The “-dis” command line “switch” tells the ASM tool to run in disassembler mode. The above command will read the “wp34s-lib.dat” binary image<sup>7</sup>, detect how many FOCAL program steps are in the one or more programs within the binary file, and generate an ASCII source listing. The ASCII listing is written to a file called “myProgs.wp34s”.

For example, suppose the flash image contained a modular normalization program<sup>8</sup>. The output listing, “myProgs.wp34s”, might look something like this:

```
// Source file(s): wp34s-lib.dat
0001 LBL'MOD'
0002 RMDR
0003 RCL L
0004 x[<->] Y
0005 x<0?
0006 +
0007 RTN
0008 END
// 8 total words used.
// 9 total words used.
// 7 single word instructions.
// 1 double word instructions.
```

**Figure 4.2: Disassembled Source Example**

Though the disassembler prints program “step numbers”, these are more for the user's benefit than anything else since they are ignored when the file is assembled. The optional step numbers can also be followed by an optional colon (“:”) when the user writes their own ASCII source file. Additionally, the LBL mnemonics can be optionally preceded by one or more asterisks.

<sup>6</sup> When exiting, the emulator writes one or more files with the name “wp34\*.dat”. All these files except the “wp34-R.dat” file may be disassembled with this script. The files named “wp34-[0-8].dat” are flash images and are the primary target file names for assembly output by this tool. See the actual **WP 34S** documentation for more (and certainly more correct!) details.

<sup>7</sup> Conventionally, the flash images have the extension “.dat”, however there is nothing that enforces this extension. The **WP 34S** emulator, however, recognizes several pre-named “.dat” files. See the main **WP 34S** documentation for more details.

<sup>8</sup> Likely the flash image will contain many more programs than this since the standard distribution usually sets up a library of default programs. However, use your imagination a bit...



## 4.2 Assembly Introduction

The same tool can also take ASCII FOCAL source files and turn them into binary images as well. The following command line may be used to convert an ASCII FOCAL program file into a binary flash image:

```
$ wp34s_asm.pl myProgs.wp34s -o wp34s-lib.dat
```

**Figure 4.3: Assembler Example Command-Line**

The resultant binary flash image, in this case “wp34s-lib.dat”, can be loaded directly into either the emulator or the calculator using instructions found in the main **WP 34S** calculator documentation.

## 4.3 Disassembler Instructions

The disassembler uses a direct lookup technique to translate the series of 16-bit words found in the binary flash image back into FOCAL source mnemonics. Most **WP 34S** op-codes are one 16-bit word – the exception being those that take a quoted alpha string. The quoted alpha strings are between 1 and 3 characters (eg: LBL 'xx', INT 'yyy', etc.). Quoted alpha op-codes take two 16-bit words to encode.

As can be seen in Figure 4.4, to invoke the disassembler the tool requires the “-dis” switch and the name of the binary image to disassemble<sup>9</sup>. If the “-o” switch is not given in disassembler mode, the output is sent to the screen (STDOUT). Because STDOUT can be redirected into a file as well, hence both invocations shown below result in virtually identical listing files.

```
$ wp34s_asm.pl wp34s-3.dat -dis > myProgs.wp34s
$ wp34s_asm.pl wp34s-3.dat -dis -o myProgs.wp34s
```

**Figure 4.4: Generic Disassembler Command-Line**

The disassembler has several optional parameters that can be used. The “-s” switch (“-s <# of asterisks>”) can be used to prepend a specified number of asterisks to the front of the LBL op-codes in the listing. These improve readability of the source, making it easier to locate labels in the listing. If a value of 0 is given for this parameter, no asterisks will be added<sup>10</sup>. The current default is 4 asterisks (equivalent to “-s 4”).

```
$ wp34s_asm.pl -dis wp34s-lib.dat -s 2 -o myProgs.wp34s
```

**Figure 4.5: Disassembler Command-Line For 2 Label Asterisks**

For example, suppose the flash image contained a modular normalization program. The output of the command in Figure 4.5 might look something like this:

---

<sup>9</sup> It should be noted that the order of the command line switches is not important to the Assembler Tool Suite. Only those switches that take an extra argument are required to be in sequence.

<sup>10</sup> Asterisks in front of the LBL op-codes are ignored by the assembler.

```
// Source file(s): wp34s-lib.dat
0001 ****LBL'MOD'
0002 RMDR
0003 RCL L
0004 x[<->]y
0005 x<0?
0006 +
0007 RTN
0008 END
// 8 total instructions used.
// 9 total words used.
// 7 single word instructions.
// 1 double word instructions.
```

**Figure 4.6: Source Example with Label Asterisks**

Additionally, the disassembler can be asked to suppress the the step numbers as well by adding the “-ns” switch. This is sometimes useful when doing text comparisons<sup>11</sup> with previous listings using 'diff-ing' tools<sup>12</sup>.

```
$ wp34s_asm.pl -dis wp34s-lib.dat -ns > myProgs.wp34s
```

**Figure 4.7: Disassembler Command-Line With Step Numbers Suppressed**

The above command will produce an output listing that might look something like Figure 4.8.

```
// Source file(s): wp34s-lib.dat
LBL'MOD'
RMDR
RCL L
x[<->]y
x<0?
+
RTN
END
// 8 total instructions used.
// 9 total words used.
// 7 single word instructions.
// 1 double word instructions.
```

**Figure 4.8: Source Example with No Line Numbers**

The disassembler prints a set of diagnostic statistics at the end of the listing which includes the total number of program steps used, the number of single word instructions, and the number of double instructions. Other items might appear (or disappear!) as the tool evolves.

Note: Though the disassembler can, and does, print program step numbers, these are only for the user's benefit as they are ignored entirely by the assembler.

#### 4.4 Assembler Instructions

The assembler takes one or more ASCII FOCAL source listings and translates them into a single binary image. The assembler can be used to “mix and match” various smaller programs into a single flash image creating a “library management” system of sorts.

The assembler enforces the **WP 34S** calculator's maximum word limit in force for any given F/W version<sup>13</sup>. The user must keep the total length of any concatenated set of programs in mind when combining source programs. The statistics printed at the bottom of the disassembly listings can be

<sup>11</sup> With step numbers present, if a line is inserted, all lines below that line will show up as differences.

However, with step numbers suppressed, the one inserted line will show up clearly as the only difference.

<sup>12</sup> Examples of differencing tools include 'diff', 'tkdiff', 'meld', etc.

<sup>13</sup> This maximum quantity fluctuates with **WP 34S** F/W revision so the user is advised to review the current main documentation occasionally. The actual value is communicated to the tool suite through the wp34s.op file.

useful for tracking this. If the maximum size is exceeded, the assembler will abort with an appropriate error message. If this occurs, the user must re-balance their assembly source file and/or their mix of assembly source files to ensure that the limit is not exceeded.

*Hint: Flash programs can access alpha labels in RAM, and vice versa. Therefore a program or set of programs can be created that use XEQ or GT0 to execute programs in other areas of the calculator. This can be used to increase the effective program size that can be executed.*

When assembling a source file, the assembler **requires** that the output flash image filename be named using the “-o” switch<sup>14</sup>. Since the assembler output is binary, no option was provided to use STDOUT redirection to a file in this mode:

```
$ wp34s_asm.pl myProgs.wp34s -o wp34s-lib.dat
// Opcode map source: wp34s.op (local directory)
// Opcode SVN version: 2441
// WP 34s version: 30
// CRC16: 520A
// Total words: 32
// Total steps: 31
```

**Figure 4.9: Generic Assembler Command-Line**

After a successful assembly run, the tool will print statistics to the console for the resultant flash image. This will include the CRC16 value, the total words consumed, and the number of steps in the program. Again, the actual output may differ slightly from that shown in Figure 4.9 as the features of the tool evolve.

As with the disassembler, there are a number of options that can be used when assembling a FOCAL source.

C-style comments can be included in the user's FOCAL source. Both single line (“//”) and multi-line comment (“/\* ... \*/”) styles are supported.

```
/* Perform modular reduction with normalization: r = a mod m
   inputs: Y <= a
           X <= m
   output: X => r, such that (0 <= r < m)
*/
***LBL 'MOD'
RMDR
RCL L
x[<->]y
x<0?
+      // If negative, add the modulus back in to normalize
RTN
END
```

**Figure 4.10: Assembler Source With Comment Styles**

The FOCAL fragment shown in Figure 4.10 will produce an identical binary image to the ones in the previous examples.

As mentioned in the Disassembler section, though the disassembler can write out program step numbers in its output, these are neither required nor translated during the assembly process<sup>15</sup>.

<sup>14</sup> The assembler output is a binary file and is not suitable for displaying on the screen with a normal text editor. If the user wishes to see the actual values in this binary file, there are many programs available for this. One common one is:

```
$ hexdump wp34s-3.dat
```

<sup>15</sup> It does not matter if some statements include step numbers and some don't, nor does it matter if one or more step numbers are repeated, nor that they are not sequential (nor monotonic, blah, blah, blah). They are just plain ignored during the assembly process.

The case and format of the mnemonic **must be exact** with respect to the approved **WP 34S** syntax. See Generating Reference Op-code List on page 27 for more details on how to generate a mnemonic syntax guide table.

```
$ wp34s_asm.pl gc.wp34s fp.wp34s mod.wp34s -o wp34s-lib.dat
```

***Figure 4.11: Assembling Multiple Source Files to One Image***

The assembler can be used to concatenate several FOCAL source files into a single binary image with a single invocation. Additionally, more than one program unit can be contained within each source file. The only limitation is that the total number of words<sup>16</sup> cannot exceed the current maximum for the **WP 34S**. Figure 4.11 shows an example of using the tool to translate multiple source files into a single binary image. This technique can be useful for maintaining a library of source files by mixing and matching various programs and files into a binary image as required.

---

<sup>16</sup> Most **WP 34S** instructions occupy one 16-bit word. However there are a few classes of instruction that take 2 words. Therefore, the maximum word limit does not actually equate to the number of program steps the region can contain. It depends on the instruction mix used.

## 5 SYMBOLIC PREPROCESSOR

An optional symbolic preprocessor script (**WP 34S** PP) is available to simplify the construction of some features of the **WP 34S** calculator source files. **WP 34S** PP can either be run stand-alone with the user feeding the resultant output file into the **WP 34S** Assembler themselves, or **WP 34S** PP can be run automatically from within the **WP 34S** Assembler by adding the '-pp' command line switch to the **WP 34S** Assembler run. In the case of the latter, the assembler will automatically 'spawn' a **WP 34S** PP job and read in the post-processed source.

Currently, **WP 34S** PP has 3 main features:

- The **WP 34S** PP can use a branch pseudo-instruction that will take a 'soft' symbolic label as a target instead of a 'hard' number of steps (for BACK/STEP) or GTO/LBL. **WP 34S** PP will resolve these JMP pseudo-instructions into the appropriate instruction for the situation and will calculate and insert the correct offset when required. When a BACK/STEP is out-of-range for the required offset, **WP 34S** PP will revert to injecting a LBL, when required, and replacing the JMP with a GTO. Similarly, the GSB pseudo-instruction can be used instead of BSRV, BSRF, or XEQ/LBL in an exactly analogous manner.
- Any instruction capable of using either a numeric or single letter alpha LBL as a target can now use a 'soft' symbolic label instead. A nice byproduct is that the 'soft' symbolic label provides for some form of minimal, inherent documentation since there is no limit on the maximum label length.
- Alpha strings can be entered using a double quoted string technique rather than requiring manual decomposition into a series of "[alpha] M" or "[alpha] 'Que '" instructions.

More details for each of these topics is shown below.

### 5.1 Running the Preprocessor Stand-Alone

The **WP 34S** PP tool can be run in two ways: stand-alone with the user feeding the output to the **WP 34S** Assembler, or from within the **WP 34S** Assembler automatically.

#### 5.1.1 Stand-Alone Preprocessor

Figure 5.1 shows the **WP 34S** PP being run directly from the command line in stand-alone mode<sup>17</sup>.

```
$ wp34s_pp.pl gc_pp.wp34s mod_pp.wp34s > src.wp34s
```

**Figure 5.1: Running WP 34S PP Source Through WP 34S PP**

The resulting output file can be then fed into the **WP 34S** Assembler as described in the previous portion of this document.

#### 5.1.2 Running Preprocessor from Within Assembler

The **WP 34S** Assembler can be requested to automatically pipe the source file(s) through the **WP 34S** PP tool by adding the '-pp' switch to the command line. In this mode, the source files are automatically concatenated into a single (transient) intermediate file and a **WP 34S** PP job is spawned.

In Figure 5.2, the '-pp' switch tells the **WP 34S** Assembler to automatically feed any source files through the **WP 34S** PP tool prior to assembling them. By default, the intermediate source code is written to a file called wp34s\_pp.lst. Even though it is called a \*.lst file, this file is in the

<sup>17</sup> Throughout this document, the **WP 34S** PP source files have been given the convention of being named \*\_pp.wp34s to indicate that they contain preprocessor-type source code. This convention is not enforced nor required in any way. It is just here to help distinguish types of source files for the purpose of this document.

```
$ wp34s_asm.pl -pp gc_syn.wp34s mod_syn.wp34s -o wp34s-4.dat
```

**Figure 5.2: Running WP 34S PP Through the WP 34S Assembler**

normal \*.wp34s FOCAL source format and can be fed into the assembler at a later time if required. This file is primarily retained to give the user the ability to examine the code, or for use as a guide during debugging the program within the calculator and/or simulator.

## 5.2 JMP Pseudo Instruction

Using the native **WP 34S** source language, the user is obliged to manually count and keep track of the number of steps required for each SKIP and BACK statement in order to branch to the correct target step. If the offset to the target step exceeds the maximum number of steps allowed for these instructions (255), the BACK and/or SKIP instruction can no longer be used. If the 255 step limit is exceeded, the user must insert a LBL at the target location and use a GT0 to branch to that target step.

By replacing occurrences of BACK, SKIP, and GT0 with the single pseudo instruction JMP, and replacing any 'hard' targets with 'soft' symbolic labels, the user can make their source code easier write and to maintain. Having the ability to give the 'soft' label a meaningful name is a bonus (eg: LBL 23 vs. RelaxCoefficients:.)

Figure 5.3 is an example of the original 8-Queens benchmark program from (the **WP 34S** library) showing the original BACK and SKIP instructions highlighted in yellow.

```
0001 **LBL '8Qu'
0002 CLREG
0003 8
0004 STO 11
0005 RCL 11
0006 x=? 00
0007 SKIP 22 // branch
0008 INC 00
0009 STO[->]00
0010 INC 10
0011 RCL 00
0012 STO 09
0013 DEC 09
0014 RCL 09
0015 x=0?
0016 BACK 11 // branch
0017 RCL[->]00
0018 RCL[->]09
0019 x=0?
0020 SKIP 05 // branch
0021 ABS
0022 RCL 00
0023 RCL- 09
0024 x[!]=? Y
0025 BACK 12 // branch
0026 DSZ[->]00
0027 BACK 17 // branch
0028 DSZ 00
0029 BACK 03 // branch
0030 RCL 10
0031 RTN
0032 END
```

**Figure 5.3: Original 8-Queens Benchmark Code**

The program contains a number of branches in both the forward (SKIP) and backward (BACK) direction. When writing code such as this, the user is obliged to count the intervening steps to the desired target and insert that offset value into the instruction. In the future, if the FOCAL program is ever modified to insert or delete steps between the branch instruction and the target step, all affected branch offsets must be revisited and repaired. Furthermore, if the program is later modified to require the branch to be greater than 255 steps, the source must be modified to

replace these out-of-scope branches with GTO statements, and matching LBLs must be inserted at the desired targets. Note that the insertion of these new LBLs may drive additional SKIP/BACK branches out-of-scope as well, thus causing these newly out-of-scope branches to have to be tracked down and modified – possibly resulting in a vicious circle!

Using **WP 34S** PP, all instances of SKIP XX and BACK YY can be replaced with the unified JMP LabelZ pseudo instruction and a 'soft' symbolic label 'LabelZ: :' placed in front of the the desired target instruction. The 'soft' symbolic label at the target instruction is identified by the 2 trailing colons ("::").

The **WP 34S** PP script analyzes the target offset distance and direction and injects the appropriate instruction to execute the branch as required. If the offset is in the negative direction and less than 255, the JMP will be replaced by a BACK instruction. If the offset is positive and less than 255, the JMP will be replaced with a SKIP instruction. If the offset is greater than 255, regardless of the direction, the JMP will be replaced by a GTO with a matching numeric LBL inserted<sup>18</sup> immediately before the target step. (An example will be presented later that has the maximum offset artificially reduced to a more manageable number for example/display purposes. See Section 5.2.1 for more details.)

```

0001      *LBL'8Qu' // Entry point
0002      CLREG
0003      8
0004      STO 11
0005 loop:: RCL 11
0006      x=? 00
0007      JMP done // SKIP 22
0008      INC 00
0009      STO[->]00
0010 again:: INC 10
0011      RCL 00
0012      STO 09
0013 loop2:: DEC 09
0014      RCL 09
0015      x=0?
0016      JMP loop // BACK 11
0017      RCL[->]00
0018      RCL[->]09
0019      x=0?
0020      JMP Not0 // SKIP 05
0021      ABS
0022      RCL 00
0023      RCL- 09
0024      x[!=]? Y
0025      JMP loop2 // BACK 12
0026 Not0:: DSZ[->]00
0027      JMP again // BACK 17
0028      DSZ 00
0029      BACK 03
0030 done:: RCL 10
0031      RTN
0032      END

```

**Figure 5.4: JMP Pseudo Instruction Example – WP 34S PP Source**

To prepared the 8-Queens source for use with this the **WP 34S** PP tool, in Figure 5.4 we can see that the instances of BACK and SKIP have been replaced by the unified JMP pseudo instruction (the original BACK and SKIP instructions have been left as comments to aid in this example). The

<sup>18</sup> When replacing a JMP with a GTO, the **WP 34S** PP will opportunistically make use of any existing numeric or single letter alpha label that may meet the requirement. This prevents proliferation of 'local' labels being inserted for one logical location. **WP 34S** PP will not use quote LBLs in this manner.

symbolic labels<sup>19</sup> associated with each of these instructions are shown between the (optional) step number and the actual **WP 34S** instruction (or pseudo instruction, in the case of the Jumps).

The output of **WP 34S** PP is shown in Figure 5.5. Notice that the JMP pseudo instructions have been 'resolved' by **WP 34S** PP and, in this example, returned to the identical BACK and SKIP instructions we started with before we modified the source (nicely closing the loop on the example). As a convenience, the instructions replaced by the **WP 34S** PP tool are converted into comments for user reference<sup>20</sup>. Since this FOCAL program was so short, none of the JMP pseudo instructions were required to be 'promoted' to GT0. See Section 5.2.1 for a somewhat artificial example of JMP→GT0 promotion.

```

0001 /*      */ /* *LBL '8Qu'
0002 /*      */ /* CLREG
0003 /*      */ /* 8
0004 /*      */ /* STO 11
0005 /* loop:: */ /* RCL 11
0006 /*      */ /* x=? 00
0007 /*      */ /* SKIP 22 // JMP done
0008 /*      */ /* INC 00
0009 /*      */ /* STO[->]00
0010 /* again:: */ /* INC 10
0011 /*      */ /* RCL 00
0012 /*      */ /* STO 09
0013 /* loop2:: */ /* DEC 09
0014 /*      */ /* RCL 09
0015 /*      */ /* x=0?
0016 /*      */ /* BACK 11 // JMP loop
0017 /*      */ /* RCL[->]00
0018 /*      */ /* RCL-[->]09
0019 /*      */ /* x=0?
0020 /*      */ /* SKIP 05 // JMP Not0
0021 /*      */ /* ABS
0022 /*      */ /* RCL 00
0023 /*      */ /* RCL- 09
0024 /*      */ /* x[!=]? Y
0025 /*      */ /* BACK 12 // JMP loop2
0026 /* Not0:: */ /* DSZ[->]00
0027 /*      */ /* BACK 17 // JMP again
0028 /*      */ /* DSZ 00
0029 /*      */ /* BACK 03
0030 /* done:: */ /* RCL 10
0031 /*      */ /* RTN
0032 /*      */ /* END

```

**Figure 5.5: JMP Pseudo Instruction Example – WP 34S PP Output**

If the user's original symbolic-label source contains any 'hard' BACK and/or SKIP instructions (eg: BACK 03), these will be retained and will currently not be modified by the tool<sup>21</sup>. For example, a single BACK instruction was purposefully left in the **WP 34S** PP source to demonstrate that it is preserved. (See BACK 03 at step 029.)

<sup>19</sup> The specification for the label is that it may start with either a letter or an underscore, and may contain any combination of letters (either case), underscores, or numbers, and be terminated by a double colon. It must be at least 2 characters long (not including the trailing double colon). In terms of a regular expression, this would be: `/[A-Za-z_][A-Za-z0-9_]+:/`. Labels are case sensitive so "MyLabel0" is distinct from "myLabel0". The label must not be one that would be otherwise recognized as a 'native' label. For example, "00: :" is not allowed. However, if preceded by an underscore, "\_00: :", it is allowed. Labels must appear on a line with a valid instruction – it cannot appear on a line without a corresponding instruction.

<sup>20</sup> Note that **WP 34S** PP currently deletes all other comments as part of its synthesis process. Therefore, when debugging, it is often a good idea to have both the original \*\_pp.wp34s source and the intermediate wp34s\_pp.lst source as references. This may change in the future.

<sup>21</sup> For the moment, 'hard' BACK and SKIP instructions will be left untouched. A future version may detect these cases and convert them to 'soft' Jumps with synthetic symbolic labels inserted where required. These would then be processed in the same way as the others. This would then be useful for legacy code (if there is such a thing as legacy code for a device that is barely old enough to be out of diapers!).



### 5.2.1 JMP Target Offset Greater Than Maximum Offset Allowed

If the branch offset is greater than the maximum allowed (usually 255), the JMP pseudo instruction is replaced by a GT0 instruction and a new LBL is inserted at the target, if required.

Since this is difficult to show in a standard example due to the large program size required (by definition, more than 255 steps), the next example shows this mechanism at work but with the maximum allowed offset artificially lowered<sup>22</sup> to a value of 6 (for the purposes of this example).

The same 8-Queens benchmark output from the previous example is shown side-by-side with the maximum allowed offset of 6 version. Note the **highlighted** LBLs which were automatically inserted to reach the out-of-scope target instructions. Also note the 'promotion' of most of the SKIP and BACK instructions to GT0/LBL pairs. Observe that the JMP Not0 was converted to SKIP 05 and not to a GT0 because it remained within range of the maximum allowed offset.

As expected, the original user coded BACK 03 instruction remained untouched throughout this transformation.

|            |    |                             |            |    |                            |
|------------|----|-----------------------------|------------|----|----------------------------|
| /*         | /* | *LBL'8Qu'                   | /*         | /* | *LBL'8Qu'                  |
| /*         | /* | CLREG                       | /*         | /* | CLREG                      |
| /*         | /* | 8                           | /*         | /* | 8                          |
| /*         | /* | STO 11                      | /*         | /* | STO 11                     |
| /*         | /* |                             | /*         | /* | <b>LBL 98</b>              |
| /* loop::  | /* | RCL 11                      | /* loop::  | /* | RCL 11                     |
| /*         | /* | x=? 00                      | /*         | /* | x=? 00                     |
| /*         | /* | <b>SKIP 22</b> // JMP done  | /*         | /* | <b>GT0 99</b> // JMP done  |
| /*         | /* | INC 00                      | /*         | /* | INC 00                     |
| /*         | /* | STO [->] 00                 | /*         | /* | STO [->] 00                |
|            |    |                             | /*         | /* | <b>LBL 96</b>              |
| /* again:: | /* | INC 10                      | /* again:: | /* | INC 10                     |
| /*         | /* | RCL 00                      | /*         | /* | RCL 00                     |
| /*         | /* | STO 09                      | /*         | /* | STO 09                     |
|            |    |                             | /*         | /* | <b>LBL 97</b>              |
| /* loop2:: | /* | DEC 09                      | /* loop2:: | /* | DEC 09                     |
| /*         | /* | RCL 09                      | /*         | /* | RCL 09                     |
| /*         | /* | x=0?                        | /*         | /* | x=0?                       |
| /*         | /* | <b>BACK 11</b> // JMP loop  | /*         | /* | <b>GT0 98</b> // JMP loop  |
| /*         | /* | RCL [->] 00                 | /*         | /* | RCL [->] 00                |
| /*         | /* | RCL [->] 09                 | /*         | /* | RCL [->] 09                |
| /*         | /* | x=0?                        | /*         | /* | x=0?                       |
| /*         | /* | <b>SKIP 05</b> // JMP Not0  | /*         | /* | <b>SKIP 05</b> // JMP Not0 |
| /*         | /* | ABS                         | /*         | /* | ABS                        |
| /*         | /* | RCL 00                      | /*         | /* | RCL 00                     |
| /*         | /* | RCL- 09                     | /*         | /* | RCL- 09                    |
| /*         | /* | x[!=]? Y                    | /*         | /* | x[!=]? Y                   |
| /*         | /* | <b>BACK 12</b> // JMP loop2 | /*         | /* | <b>GT0 97</b> // JMP loop2 |
| /* Not0::  | /* | DSZ [->] 00                 | /* Not0::  | /* | DSZ [->] 00                |
| /*         | /* | <b>BACK 17</b> // JMP again | /*         | /* | <b>GT0 96</b> // JMP again |
| /*         | /* | DSZ 00                      | /*         | /* | DSZ 00                     |
| /*         | /* | BACK 03                     | /*         | /* | BACK 03                    |
|            |    |                             | /*         | /* | <b>LBL 99</b>              |
| /* done::  | /* | RCL 10                      | /* done::  | /* | RCL 10                     |
| /*         | /* | RTN                         | /*         | /* | RTN                        |
| /*         | /* | END                         | /*         | /* | END                        |

**Figure 5.6: Effect of BACK/SKIP Beyond  
Maximum Allowable Offset**

As must be reminded, this is a somewhat contrived example but is illustrative of the capability of the feature.

### 5.3 GSB Pseudo Instruction

The GSB pseudo instruction is exactly analogous to the JMP version except that it is translated to BSRB, BSRF, or XEQ/LBL. Instances of GSB whose target step is greater than 255 will be promoted to XEQ/LBL. Those that are below that offset limit will be replaced by the appropriate BSRB or BSRF instruction, depending on the sign of the offset.

<sup>22</sup> It is worth noting that to guard against certainly anomalies and corner cases, when **WP 34S** PP is invoked from within the **WP 34S** Assembler, the assembler over-rides the 255-step default and sets the maximum allowed offset to a slightly lower value instead. The reasons for this are beyond the scope of this document.

All examples for the JMP instruction are valid and equally descriptive for the GSB instruction.

Unfortunately, at time of writing, there are no examples of its use in the standard library repository<sup>23</sup>.

## 5.4 LBL-less Targets

All instructions that can take a numeric and/or single-alpha LBL can also be used with a 'soft' symbolic label with **WP 34S** PP. This includes instructions such as XEQ, GT0, etc<sup>24</sup>. If no 'hard' LBL exists at the target location identified by the matching 'soft' label, **WP 34S** PP will automatically inject a unique numeric-type LBL instruction for this purpose<sup>25</sup>. If the user's original source code contains any numeric and/or single letter LBLs, these will be made available as targets for 'soft' label instructions as well. If **WP 34S** PP is required to either branch to or reference a local LBL that already exists, it will opportunistically use the existing LBL rather than insert a new (redundant) one. Note that this is not true for quoted LBLs (ie: LBL 'ABC '). Even if a quoted LBL exists at the desired target location, **WP 34S** PP will insert a unique local numeric LBL for its own purposes unless there is also an equivalent local LBL<sup>26</sup>.

The example in Figure 5.7 shows a program fragment with an XEQ instruction 'calling' a subroutine with a purely symbolic ('soft') label at the target. Normally, the user would have added a LBL instruction to begin the 'Vsub' subroutine. In this example, the start of the subroutine is marked by only a 'soft' symbolic label; in this case 'Vsub: '.

```
Vabs::  LBL 'VAB'
        XEQ Vsub
        x[<->] T
        STO J
        DROP
        [cplx]STO L
        [cplx]DROP
        RTN

// Tool will automatically inject a LBL here
Vsub::  ENTER[^]
        x[^2]
        RCL Z
        RCL[times] T
        +
        RCL T
        RCL[times] T
        +
        [sqrt]
        RTN
        END
```

**Figure 5.7: LBL-less Example – WP 34S PP Source**

As shown in Figure 5.8, **WP 34S** PP detects this relationship and automatically inserts a LBL instruction at the appropriate location<sup>27</sup>. Additionally, the LBL number – LBL 99 in this case –

<sup>23</sup> However, GSB is currently extensively used in the XROM area of the **WP 34S** F/W. The interested reader can wade through the **WP 34S** source directories to view the XROM files – could be a daunting task!

<sup>24</sup> Besides LBL, at time of writing, the instructions that can take a 'local' label include: LBL?, XEQ, GT0, [SIGMA], [PI], SLV, f'(x), f''(x), and [integral].

<sup>25</sup> **WP 34S** PP currently does not support local label re-use. Multiple instances of numeric and/or single letter labels will cause **WP 34S** PP to issue an error and exit.

<sup>26</sup> Equivalent LBLs are defined as more than one consecutive LBL.

<sup>27</sup> The actual numeric value of the label is largely immaterial. It could be any one of the 100 available LBLs. By default, **WP 34S** PP will start at LBL 99 and will work backwards as each newly required LBL is injected. However, if the next synthetic LBL number to be injected already exists in the user's original source code, **WP 34S** PP will detect its presence and will choose the next non-colliding number instead. The astute observer will have noted that the LBL is injected one step before the symbolically labelled instruction. This is intentional and aids in possibly (re)using the symbolically labelled instruction with another SKIP instruction generated by the JMP pseudo instruction. (Moving forward, if a SKIP lands on a

was also substituted in the XEQ instruction as well (with the original source instruction moved into a comment field).

```

0001 /* Vabs:: */ LBL 'VAB'
0002 /*      */ XEQ 99 // XEQ Vsub
0003 /*      */ x[<->] T
0004 /*      */ STO J
0005 /*      */ DROP
0006 /*      */ [cmplx]STO L
0007 /*      */ [cmplx]DROP
0008 /*      */ RTN
0009 /*      */ LBL 99
0010 /* Vsub:: */ ENTER[^]
0011 /*      */ x[^2]
0012 /*      */ RCL Z
0013 /*      */ RCL[times] T
0014 /*      */ +
0015 /*      */ RCL T
0016 /*      */ RCL[times] T
0017 /*      */ +
0018 /*      */ [sqrt]
0019 /*      */ RTN
0020 /*      */ END

```

**Figure 5.8: LBL-less Example – WP 34S PP Output**

## 5.5 Double Quoted Text Strings

In the original incarnation of the **WP 34S** calculator, in order to enter multiple character alpha strings, the user was obliged to enter a long series of single character mnemonics such as shown in the top portion of Figure 5.9.

A later development has seen the addition of triple alpha strings using the quoted alpha mnemonic. Shown in the bottom portion of Figure 5.9.

```

// The original methodology for entering strings:
CL[alpha]
[alpha] W
[alpha] P
[alpha] [space]
[alpha] 3
[alpha] 4
[alpha] S
[alpha]VIEW
PSE 08

// Improved triple alpha method:
CL[alpha]
[alpha]'WP[space]'
[alpha]'34S'
[alpha]VIEW
PSE 08

```

**Figure 5.9: Original Alpha Strings Programs**

With **WP 34S** PP comes the ability to use double quoted strings of arbitrary length. **WP 34S** PP will parse the strings into optimal groups of 3 alpha character as required. When it does not have a sufficient number of characters to make up a triplet, it reverts to the single character op-code<sup>28</sup>.

As a convenience, **WP 34S** PP will also recognize embedded spaces and correctly translate these to the correct '[space]' mnemonic as well. See Figure 5.10 for an example.

LBL is effectively treated as a NOP. This has the effect of increasing the dynamic range of the SKIP offset by the number of LBLs between the SKIP offset target and the actual target instruction with the symbolic label.)

<sup>28</sup> It turns out that there is no code density difference between a 2 single [alpha] X (4 bytes) and a triplet [alpha] 'XX' with the 3<sup>rd</sup> character nulled out (4 bytes).

```
// PP double quoted methodology for entering strings:  
CL[alpha]  
"WP 34S"  
[alpha]VIEW  
PSE 08
```

**Figure 5.10: Double Quoted Alpha String Example**

All three program fragments will result in exactly the same text in the **WP 34S** alpha display. In the case of Figure 5.10, the disassembly of such a program is shown below. It can be seen to be essentially identical to the program in the lower half of Figure 5.9.

```
0001 CL[alpha]  
0002 [alpha]'WP[space]'  
0003 [alpha]'34S'  
0004 [alpha]VIEW  
0005 PSE 08  
// 5 total instructions used.  
// 7 total words used.  
// 3 single word instructions.  
// 2 double word instructions.
```

**Figure 5.11: Doubled Quoted Alpha String Disassembly**

## 6 LIBRARY MANAGER

The library manager (LIB) is designed to help the user deal with binary libraries. It allows user to extract a catalogue of programs contained in the binary image, add new programs to the image, remove existing programs, replace or update existing programs, as well as a few other things.

Under the hood, the tool works by launching one or more ASM 'jobs' to disassemble the original binary image back into a temporary ASCII FOCAL file. This is then split into individual program segments delimited by a quoted text LBL and an END statement<sup>29</sup>.

This list of program segments is then processed as per the requested operations (eg: remove a program called 'TNC', add a program named 'JNK', replace the existing program named 'TWA' with its new FOCAL source, etc.). Finally, LIB converts the resulting temporary FOCAL program group back into a binary image by automatically launching the ASM and PP tools again. The final binary image can either overwrite the original binary file or can be written to a new one, as directed by the command line options.

The LIB tool can process both the flash-type and the emulator state-type binary images<sup>30</sup>. It can even convert from one style to the other<sup>31</sup>.

### 6.1 Library Manager Options

In most cases, the LIB tool reads in an existing binary library and processes it in some way. The source library is named via the `"-ilib someSrcLib.dat"` switch. By default, the output must go to another file named by the `"-olib someDstLib.dat"` switch. However, the LIB tool can be requested to overwrite the input library file using the `"-f"` switch – forcing an overwrite.

There is one exception to the general rule of requiring an input library name and that is when "adding" a bunch of new programs and creating a new flash library. In this case, LIB basically acts in a manner similar to the ASM tool and generates a binary library from scratch. Note that this 'brand new' output library can only be generated for flash-type library. Emulator state-type libraries cannot be generate this way because they need a 'seed' state library<sup>32</sup>.

A catalogue can be generated of the program names within the library using the `"-cat"` switch.

One or more programs can be added to a library by simply naming them on the command line. The same syntax is used when replacing an existing program within a library as well.

One or more programs can be removed from a library by naming them with the `"-rm"` switch. This switch can be used multiple times and/or multiple program names can be given by delimiting with double quotes.

### 6.2 Library Manager Examples

The following is a tutorial of the basic operations available to the LIB program.

---

<sup>29</sup> A consequence of this LBL-END pair requirement is that the, if a disassembled program or source program is discovered with no END statement, one will be automatically inserted at the end (go figure!). This may or may not be where the user intended the missing instruction to go – *caveat emptor!* Best to create valid source in the first place!!

<sup>30</sup> The internal format of these two files is different. The differences are mostly to do with where the CRC field is located and what portion of the file is covered by the CRC. The two formats are not 100% compatible!

<sup>31</sup> Converting from an emulator state file to a flash image is relatively straight forward and can be done with no additional information. Converting from a flash image to a emulator state file is more *interesting*. In order to either create a state file or convert a flash image to a state file, the user must supply a seed state file previously generated by the emulator. This is as simple as starting the emulator and exiting it. Upon exiting, the emulator will save out a state file (wp34s.dat). This file can be given to the LIB tool as its seed.

<sup>32</sup> The 'seed' state image is required because the the complexity of the state file contents. The content outside the program area is beyond the scope of this tool.

### 6.2.1 Generating a Catalogue

The following example shows how to generate a catalogue of programs contained in a binary library. Note that since we are not asking for any modifications to the library (just a catalogue), thank you), we do not need to specify an output library name (or a “-f” overwrite command).

Using the “-cat” command might result in something like Figure 6.1, indicating there are 4 programs in this library: 'MOD', '8Qu', 'CBK', and 'Roc'. The first begins at step 1, the second at step 9, and so on.<sup>33</sup>

```
$ wp34s_lib.pl -cat -ilib wp34s-lib.dat
Initial library catalogue (format: flash file)
  Source: wp34s-lib.dat, Program name: MOD, Line number: 1
  Source: wp34s-lib.dat, Program name: 8Qu, Line number: 9
  Source: wp34s-lib.dat, Program name: CBK, Line number: 40
  Source: wp34s-lib.dat, Program name: Roc, Line number: 106
Library details:
// WP 34S assembly preprocessor enabled: '-pp'
// Opcode map source: wp34s.op (local directory)
// Opcode SVN version: 2441
// Running WP 34S preprocessor from: wp34s_pp.pl
// WP 34s version: 30
// CRC16: 6E9D
// Running in V3 Flash-mode. Max words: 9999
// Total words: 132
// Total steps: 119
```

**Figure 6.1: Generating a Catalogue from a Binary Library**

### 6.2.2 Deleting One or More Programs

Using the catalogue, and thus knowing the names of the programs within the library, the following example shows how to delete one or more programs from a library. Note that we must specify either an output library filename or use the “-f” switch to force the input library file to be overwritten. In this case we named a new one using the “-olib” switch.

Here are removing both 'MOD' and 'CBK' from the library.

---

<sup>33</sup> A bit of arithmetic can even tell you how many lines each are: 'MOD' is 9-1 or 8 steps, '8Qu' is 40-9 or 31 steps, 'CBK' is 106-40 or 66 steps, and 'Roc' is 132-106 or 26 steps. I realize the tool should generate this extra info for you and it may get there yet!

```

$ wp34s_lib.pl -cat -ilib wp34s-lib.dat -olib new.dat \
  -rm "CBK MOD"
Initial library catalogue (format: flash file)
  Source: wp34s-lib.dat, Program name: MOD, Line number: 1
  Source: wp34s-lib.dat, Program name: 8Qu, Line number: 9
  Source: wp34s-lib.dat, Program name: CBK, Line number: 40
  Source: wp34s-lib.dat, Program name: Roc, Line number: 106
Removing program: "CBK", old program steps: 66
Removing program: "MOD", old program steps: 8
Modified library catalogue (format: flash file)
  Source: new.dat, Program name: 8Qu, Line number: 1
  Source: new.dat, Program name: Roc, Line number: 32
Library details:
// WP 34S assembly preprocessor enabled: '-pp'
// Opcode map source: wp34s.op (local directory)
// Opcode SVN version: 2441
// Running WP 34S preprocessor from: wp34s_pp.pl
// WP 34s version: 30
// CRC16: FBF3
// Running in V3 Flash-mode. Max words: 9999
// Total words: 51
// Total steps: 45

```

**Figure 6.2: Deleting Programs from Binary Library**

Based on the lengths we calculated from the previous example, we can confirm that the new library size ( $119 - 8 - 66 = 45$ ) is correct.

The program names to be removed can also be added as individual “-rm” switches as well – or using both techniques at once if you wish. Here is an example that results in an identical output library as the previous invocation (notice that the catalogue has been suppressed this time resulting in a reduced verbosity transaction listing). We have also forced the output library to be written over top the input library by using the “-f” switch.

```

$ wp34s_lib.pl -ilib wp34s-lib.dat -f -rm CBK -rm MOD
Removing program: "CBK", old program steps: 66
Removing program: "MOD", old program steps: 8
Library details:
// WP 34S assembly preprocessor enabled: '-pp'
// Opcode map source: wp34s.op (local directory)
// Opcode SVN version: 2441
// Running WP 34S preprocessor from: wp34s_pp.pl
// WP 34s version: 30
// CRC16: FBF3
// Running in V3 Flash-mode. Max words: 9999
// Total words: 51
// Total steps: 45

```

**Figure 6.3: Deleting Programs from Binary Library – Alternate**

### 6.2.3 Adding or Replacing One or More Programs

You can add new or replace existing programs in a library as well. New programs with the same name as existing programs will result in the original programs being replaced. Again, since we are asking the library to be modified, we need to specify something about the output library. In this case we named a new one using the “-olib” switch.

In this example we are adding a new program (from file “sum.wp34s”) and replacing an existing one (program ‘8Qu’ from file “8queens\_pp.wp34s”).

```
$ wp34s_lib.pl -ilib wp34s-lib.dat -olib new.dat \
    sum.wp34s 8queens_pp.wp34s
Replacing program: "8Qu", old program steps: 31, new program steps: 31
Adding program: "[sigma][SIGMA][DELTA]", new program steps: 44
Library details:
// WP 34S assembly preprocessor enabled: '-pp'
// Opcode map source: wp34s.op (local directory)
// Opcode SVN version: 2441
// Running WP 34S preprocessor from: wp34s_pp.pl
// WP 34s version: 30
// CRC16: AA53
// Running in V3 Flash-mode. Max words: 9999
// Total words: 96
// Total steps: 89
```

**Figure 6.4: Adding and Replacing Programs from Binary Library**

Note, however, if we had instead added the non-PP version of the 8-Queens program from the library directory in the SVN repository, since they actually have different names ('8Qu' vs. '8QU'), the new non-PP version would have been added rather than replacing the PP version.

In Figure 6.5, we can see that the catalogue reports that both '8Qu' and '8QU' exist and they are separate and unique programs to the calculator.

```
$ wp34s_lib.pl -ilib wp34s-lib.dat -olib new.dat \
    sum.wp34s 8queens.wp34s -cat
Initial library catalogue (format: flash file)
  Source: wp34s-lib.dat, Program name: 8Qu, Line number: 1
  Source: wp34s-lib.dat, Program name: Roc, Line number: 32
Adding program: "8QU", new program steps: 31
Adding program: "[sigma][SIGMA][DELTA]", new program steps: 44
Modified library catalogue (format: flash file)
  Source: new.dat, Program name: 8QU, Line number: 1
  Source: new.dat, Program name: 8Qu, Line number: 32
  Source: new.dat, Program name: Roc, Line number: 63
  Source: new.dat, Program name: [sigma][SIGMA][DELTA], Line number: 77
Library details:
// WP 34S assembly preprocessor enabled: '-pp'
// Opcode map source: wp34s.op (local directory)
// Opcode SVN version: 2441
// Running WP 34S preprocessor from: wp34s_pp.pl
// WP 34s version: 30
// CRC16: E95E
// Running in V3 Flash-mode. Max words: 9999
// Total words: 128
// Total steps: 120
```

**Figure 6.5: Adding Programs from Binary Library**

#### 6.2.4 Converting Binary Libraries to Flash Formats

As has been mentioned, there are 2 formats of binary libraries use in the **WP 34S** calculator and emulator. The first format is the state file which is the RAM and register state. The other format is the flash image. For the purposes of these tools, they mostly differ in the specification of the CRC (where it is locate and what it covers) and how many steps the image may hold (flash is much larger).

The following converts a state library to a flash format (using the “-conv” switch).



```
$ wp34s_lib.pl -ilib wp34s.dat -olib flash.dat -cat -conv
Initial library catalogue (format: State file)
  Source: wp34s.dat, Program name: 8Qu, Line number: 1
  Source: wp34s.dat, Program name: Roc, Line number: 32
Modified library catalogue (format: flash file)
  Source: flash.dat, Program name: 8Qu, Line number: 1
  Source: flash.dat, Program name: Roc, Line number: 32
Library details:
// WP 34S assembly preprocessor enabled: '-pp'
// Opcode map source: wp34s.op (local directory)
// Opcode SVN version: 2441
// Running WP 34S preprocessor from: wp34s_pp.pl
// WP 34s version: 30
// CRC16: FBF3
// Running in V3 Flash-mode. Max words: 9999
// Total words: 51
// Total steps: 45
```

**Figure 6.6: Converting Library from State to Flash Format**

Notice that the transaction record indicates that the original format was state<sup>34</sup> and the final format is flash. All examples up until Figure 6.6 have assumed that both the input and output libraries were in flash format (the default unless otherwise told). The ' -conv ' switch assumes that the input is state and the output is flash.

### 6.2.5 Dealing with State File Libraries

As has been mentioned, the state file format is different from the flash format<sup>35</sup>. The suite of commands shown in most of the previous examples (except the “-conv” which is already assumed to be state file format) can also be used with a “-state” switch in order to indicate that the output file is to be generated in state file format.

This switch can be used to operate on a state file (ie: state file in → state file out) or can be used to convert a flash to a state file<sup>36</sup> (ie: flash file in → state file out).

```
$ wp34s_lib.pl -ilib wp34s.dat -cat -state -olib state.dat mod.wp34s
Initial library catalogue (format: State file)
  Source: wp34s.dat, Program name: MNO, Line number: 1
Adding program: "MOD", new program steps: 8
Modified library catalogue (format: State file)
  Source: state.dat, Program name: MNO, Line number: 1
  Source: state.dat, Program name: MOD, Line number: 10
Library details:
// WP 34S assembly preprocessor enabled: '-pp'
// Opcode map source: wp34s.op (local directory)
// Opcode SVN version: 2441
// Running WP 34S preprocessor from: wp34s_pp.pl
// WP 34s version: 30
// CRC16: 61FF
// Running in V3 State-mode. Max words: 922
// Total words: 19
// Total steps: 17
```

**Figure 6.7: Operating on a State File**

Notice that the transaction record indicates several items of interest. Firstly, since the “-state” switch was used, the tool changes to outputting a state file<sup>37</sup>. Secondly, we see that the “max words” limit is set at 922. The RAM area for FOCAL programs is much smaller than the flash area!

<sup>34</sup> It is important to note that the tool itself cannot automatically distinguish between the two formats but must be told which format is which.

<sup>35</sup> It turns out that the file format only matters when the binary image is written out. Only at that time is it actually required to know whether the format is a state file or a flash file.

<sup>36</sup> The flash → state translation comes about because the above reason – the tool doesn't actually care whether the input is state or flash. Knowing the format only matters when writing the output file.

<sup>37</sup> Though the input is shown to be state format as well, this does not actually make any difference. For disassembly purposes, the format does not matter.

```
$ wp34s_lib.pl -ilib wp34s.dat -cat -olib flash.dat mod.wp34s
Initial library catalogue (format: flash file)
  Source: wp34s.dat, Program name: MNO, Line number: 1
Adding program: "MOD", new program steps: 8
Modified library catalogue (format: flash file)
  Source: flash.dat, Program name: MNO, Line number: 1
  Source: flash.dat, Program name: MOD, Line number: 10
Library details:
// WP 34S assembly preprocessor enabled: '-pp'
// Opcode map source: wp34s.op (local directory)
// Opcode SVN version: 2441
// Running WP 34S preprocessor from: wp34s_pp.pl
// WP 34s version: 30
// CRC16: 61FF
// Running in V3 Flash-mode. Max words: 9999
// Total words: 19
// Total steps: 17
```

**Figure 6.8: Converting State to Flash Format**

Operating on the same file (which is in fact a state file!), we can convert it to a flash format by **not** specifying that it is a state file. Without other command line switches, the tool will assume that the output format is the default flash image mode and will write the binary image accordingly. Note that currently we actually have to do something to the file such as add or remove a program in order to get an output to be written (this may change in the future).

As well, notice that since this is a flash format, the “max words” limit now is set to a whopping 9999 words. The flash area is much larger than the state RAM area.

## 7 ADDITIONAL INFORMATION

### 7.1 Generating Reference Op-code List

In order to make it easier to understand the current **WP 34S** op-code mnemonics, the **WP 34S** Assembler Tool Suite script can create a reference file of legal op-codes for use as a guideline by using the following command:

```
$ wp34s_asm.pl -syntax legal_opcodes.lst
```

This will produce a rather sizable list of all op-codes<sup>38</sup> the **WP 34S** recognizes within FOCAL programs.

As the **WP 34S** evolves, this list may change from time to time. It may be prudent to regenerate the list at intervals as the project progresses<sup>39</sup>.

Figure 7.1 shows a fragment of a generate syntax-helper file<sup>40</sup> (first and last 5 lines).

```
// Opcode SVN version: 2441
0000 ENTER[^]
0002 EEX
0003 +/-
0004 .
0005 0
...
f600 SLV'
f700 f'(x)'
f800 f"(x)'"
f900 [integral]'
fa00 [alpha]'
```

**Figure 7.1: Abridged View of Op-Code Syntax Table**

The first field is the hexadecimal value of the op-code used by the **WP 34S**. The next field(s) are the mnemonic used in the listing. Note that currently the assembler is not very forgiving of the format of the mnemonic field; it must match the format in this file exactly. Where there is whitespace<sup>41</sup> within the mnemonic, the user must retain the exact whitespace in their FOCAL source files. Many special printing characters are *escaped* by the use of square braces (“[ . . . ]”). These must also be faithfully reproduced<sup>42</sup>.

### 7.2 Migrating Programs Between Op-Code Revisions

From time to time, the **WP 34S** op-code definitions may change because of the addition or deletion of certain functions and/or constants. The combination of being able to extract the flash image, disassemble the program to an ASCII source format using a specific op-code definition map, and then reassemble it into the new op-code definition map is a useful feature of the **WP 34S** Assembler Tool Suite tool.

Using SVN<sup>43</sup> it is possible to generate an op-code definition map for the **WP 34S** calculator F/W revision in use when the FOCAL program was originally keyed into the calculator or generated by

<sup>38</sup> At time of writing, this syntax table is greater than 500KB in size!

<sup>39</sup> A helpful technique is to name the syntax guide file after the SVN revision number of the script source. For example, if the script's SVN number is 2441, a suggestion is to name the syntax guide file “syntax\_2441.lst”.

<sup>40</sup> As can be seen, this file is from the SVN 2441 version and may bear little resemblance to the current version. It is shown to illustrate the table format.

<sup>41</sup> “Whitespace” is a common programming term used to mean one or more contiguous characters that show as spaces. For the purposes of this usage, it is taken to mean one or more tabs and/or spaces in a row.

<sup>42</sup> A planned evolution of the Assembler Tool Suite project is to make this mnemonic format somewhat more “friendly”. However, until then, it is very strict in its mnemonic format.

<sup>43</sup> It is expected that the user can look up SVN usage via the usual Internet resources so only very bear commands will be presented here – without much explanation.

this tool suite. This op-code definition map can be saved and used to disassemble the user's program<sup>44</sup> at a later time.

```
$ wp34s_asm.pl -dis wp34s-lib.dat -opcode wp34s_XXXX.op > mySrc.wp34s
```

**Figure 7.2: Command-Line to Disassemble Using a Specified Op-Code Map**

Repeat the last step for as many binary images as are required. Place each output listing in a separate “\*.wp34s” file.

With the working user program(s) safely translated to an ASCII FOCAL source format, the **WP 34S** development branch can be brought up to the head of the SVN tree. The updated op-code map and re-assembly is shown below:

```
$ wp34s_asm.pl mySrc.wp34s -opcode wp34s_YYYY.op -o wp34s-lib.dat
```

**Figure 7.3: Command-Line to Assemble Using a Specified Op-Code Map**

The calculator can now be re-flashed with the latest “./trunk/realbuild/calc.bin” F/W image and the newly translated flash user programs restored as per the instructions in the main **WP 34S** calculator documentation.

Equally, this technique may be used to move a modern program to an older **WP 34S** revision<sup>45</sup>. The user must be careful no new op-codes, constants, and/or conversion factors were used in the modern program when regressing to previous SVN versions. Unrecognized op-codes will cause the assembler to halt and display an appropriate error.

### 7.3 Automatically Locating Op-code Map

The suite uses an external op-code translation table generated during the **WP 34S** development build process (wp34s.op). This file can be either specified or automatically “discovered” by the script.

The script decides which op-code map to use in the following order:

1. If the user has referenced a map file via the command line switch (ie: -opcodes wp34s\_1234.op) this map will be used.
2. If the above is not the case, the script looks in the current directory for a file called wp34s.op.
3. If the local file is not found, the script looks in the directory where the script itself is located for a file called wp34s.op.
4. If none of the above is available, the script issues an appropriate error and exits.

---

<sup>44</sup> The Linux command set is presented here. The author has never done these steps on any other operating systems though they *should* be straight forward – provided the library dependencies are satisfied.

<sup>45</sup> Moving modern programs to older **WP 34S** releases is seen as very unlikely to happen since the majority of calculators will tend to be re-flashed in the *forward* direction (ie: higher SVN revision numbers).

## 8 EXAMPLES LIBRARY

The “./library/” directory distributed with the **WP 34S** project includes a number of **WP 34S** FOCAL programs contributed by users. A secondary purpose of these programs is to be instructive in the use of the **WP 34S** assembler source language.

Several variations of source format are presented in the collection of files. These include:

- The various comment formats
- The optional preceding step number
- The optional asterisk(s) on labels

The user is encouraged to use these programs as examples for how to construct ASCII FOCAL source files for the **WP 34S**. A README\_ASM file is included in the directory with more information.

## 9 SOURCE OF PERL PACKAGES

Perl is a platform independent interpreted scripting language. Perl interpreters are available for many different O/S platforms.

### 9.1 Linux

A Perl interpreter is included with virtually every Linux distribution available<sup>46</sup>. With Linux, there is usually nothing more that needs to be done beyond running the examples as shown in the preceding sections.

### 9.2 Windows

Perl must be added separately to a Windows system. Fortunately there are several very good Perl packages available for Windows<sup>47</sup>.

There are three free packages that are particularly recommended – none above the other, and in no particular order.

#### 9.2.1 Cygwin

Cygwin is a package intended to provide many of the Linux command utilities in a Windows environment. It is available from the <http://cygwin.org/> website. Installation of cygwin is (well) beyond the scope of this document – and is sometimes not trivial to complete – but is well worth the effort if you wish to enjoy a wide variety of Linux-like commands.

Once installed, it is best to open a command shell (bash, zsh, csh, etc.) and “fill your boots”<sup>48</sup>. If you are well versed in Linux environments, cygwin will be immediately familiar.

#### 9.2.2 Strawberry Perl

Strawberry Perl is a Perl package complete with a full collection of Perl libraries. It is available as a Windows install package from the <http://strawberryperl.com/> website.

Since this package's installer does not make the pathext associations during installation, in order to run a Perl script, the user must “give the script” to the Perl interpreter for it to be run. For example, Figure 9.1 shows how this is done:

```
$ perl ..\trunk\tools\wp34s_asm.pl -dis wp34s-3.dat > mySrc.wp34s
```

**Figure 9.1: Running Script Under Strawberry Perl**

The Strawberry Perl installer does not setup the file associations for Perl scripts (as Active State does). There are many good reference for “correcting” this oversight<sup>49</sup>.

#### 9.2.3 Active State Perl

Active State Perl is another free Perl package distributed by a commercial venture that also supplies a licensed enterprise version. This is available from the <http://www.activestate.com/activeperl> website.

Active State Perl has a slight advantage over Strawberry Perl in that its installation script will set appropriate associations to the '.pl' extension for the Perl interpreter, making the command shell automatically recognize Perl scripts and launch the interpreter accordingly. Hence, other than the

<sup>46</sup> Actually, the author is unaware of any that does *not* include a Perl package.

<sup>47</sup> And some very poor packages as well (*caveat utilitor!*).

<sup>48</sup> It is not the intention of this document to teach the finer points of cygwin usage. Consult one of the many fine on-line documents for further information.

<sup>49</sup> See <http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/ftype.mspx> for an example of how this may be done.

fact that the path names must be “Window-ized”, the examples shown in the first part of this manual are equally valid.

```
$ ..\trunk\tools\wp34s_asm.pl -dis wp34s-3.dat > mySrc.wp34s
```

**Figure 9.2: Running Script Under Active State Perl**

#### 9.2.4 Native Windows Executable

There is an experimental Windows stand-alone EXE file built using a Perl-to-executable module called PAR::Packer<sup>50</sup>. PAR::Packer includes a tool called “pp” which “packages” the Perl script along with Perl libraries into an executable form<sup>51</sup>.

This Assembler Tool Suite has been packaged using the “pp” tool into a Windows executable.

```
$ wp34s_asm.exe -dis wp34s-3.dat -o mySrc.wp34s # disassemble
$ wp34s_asm.exe mySrc.wp34s -o wp34s-3.dat # assemble
```

**Figure 9.3: Running the Windows Executable**

The executable should be usable on virtually any Windows installation with no additional libraries or files required<sup>52</sup>. It has exactly the same command line usage as the base Perl script – albeit with the requirement to “Window-ized” the directory paths.

A few words on the “pp” tool and its output:

1. The executable image is somewhat on the *large* size because of the way “pp” lumps into the EXE many libraries which may not be strictly required. A quick stab at reducing the footprint has been attempted (this succeeded in reducing the “static” executable footprint from ~12.5MB down to ~5.6MB!). If time permits, more work will be done to further reduce the footprint.
2. Since “pp” constructs a compressed image of the Perl libraries, the first invocation will take some time to decompress the image. Subsequent runs should benefit from the fact that “pp” mysteriously caches the decompressed image (don't ask me how, don't ask me where!).
3. Since the decompressed image must exist for the script to run, the “dynamic” footprint (ie: the decompressed directory) will be somewhat larger than the static executable. The author is not sure exactly how large this is but will estimate in the 20MB+ range (purely a guess!).

### 9.3 MAC O/S

The author is under the impression that MAC O/S comes with a Perl interpreter. However, he is not actually very familiar with the MAC. If someone would care to update this section, it would be appreciated.

<sup>50</sup> Available from CPAN. Current version v1.009.

<sup>51</sup> At time of writing, the native executable version of the script was built using the aforementioned version of PAR::Packer and Strawberry Perl, version 5.12.1.0.

<sup>52</sup> The executable version of the script will also automatically locate the wp34s.op file, if available, as described in Section 7.3.

## 10 HELP

All scripts from the **WP 34S** Assembler Tool Suite have help screens which explain the command-line arguments that can be used. Use the '-h' switch to display the help screen.

```
$ wp34s_asm.pl -h
$ wp34s_pp.pl -h
$ wp34s_lib.pl -h
```

*Figure 10.1: Invoking the HELP Screens*