WP 34s Assembler/Disassembler User Guide

Table of Contents

Table of Figures

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.

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) scientific programmable calculators (see http://sourceforge.net/projects/wp34s/ for more details). The ***WP 34S*** project was done with the knowledge and occasional assistance of HP staff.

Most programmable calculators have their user programs stored in RAM but, in these modern days, the RAM is backed up by the battery even when the calculator is "off". The ***WP 34S*** has the additional capability of storing user programs in one of 9[1] flash memory pages.

This assembler/disassembler program is targeted at helping to design, archive, distribute, and maintain those ***WP 34S*** user programs.

1.1 Goals

The ***WP 34S*** Assembler/Disassembler was designed to accomplish several goals:

1. Archiving and distribution of ***WP 34S*** 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 user programs to be saved in an ASCII source format, they can be more easily ported between revisions.
3. Calculator programmers often develop a suite of 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.

Thus was born the ***WP 34S*** Assembler/Disassembler.

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[2] shows SVN 1173 does not indicate that this is the last version supported.

1.3 Acknowledgements

Without the tireless efforts of Walter Bonin, Paul Dale, and Marcus von Cube (the "Gang of Three"[3]) 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

[1] The number of flash pages available to the ***WP 34S*** is dependent on the calculator's firmware load. As the capability evolves, this number changes. At time of writing (SVN 1446) this currently stands at 9, but may change in the future – sometimes without notice. Consult the latest documentation for the calculator for up-to-date information.

[2] The cover image was shamelessly stolen from the main ***WP 34S*** documentation. 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.

[3] Thanks to Gerry Schultz for coming up with a better title than "techo-trinity".

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. And finally, Bruce Bergman, who has built a web interface[4] to allow anyone to use this tool, regardless of whether they know what a command line application is or not.

[4] See http://www.wiki4hp.com/doku.php?id=34s:assembler

2 LIST OF ACRONYMS AND DEFINITIONS

CPAN	Comprehensive Perl Archive Network (http://www.cpan.org)
dat	Filename extension of a binary program image from flash or RAM used by the emulator
F/W	Firmware
PP	***WP 34S*** Symbolic Preprocessor
RPN	Reverse Polish Notation
SDK	Software Development Kit
SVN	Subversion – a revision control tool
S/W	Software
wp34s	Filename extension of a ***WP 34S*** source file (by convention)

3 OVERVIEW

The ***WP 34S*** Assembler/Disassembler uses the op-code format as described by the op-code list generated by the "`calc`" program. See Section 5.1 Generating Reference Op-code List for more details.

The Assembler/Disassembler is currently a single pass, direct translation tool. An optional Preprocessor front-end is also available to convert some symbolic representations into 'pure' code that can be translated by the Assembler.

The Assembler/Disassembler tool chain is written entirely in Perl (version 5.8+) so as to be portable across many platforms. However, 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 5.3 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 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 source files.
- Will automatically locate the most current op-code map if updated via SVN.
- An optional symbolic preprocessor script can be used to simplify ***WP 34S*** source code.

4 USAGE

This tool is both an assembler and a disassembler in one script. For most people, the disassembler will be the first mode they will use – and this will likely be to disassemble a flash image recovered directly from the ***WP 34S*** calculator and/or the emulator.

The disassembler reads a binary image created with the "SAVE", "PSTO", "P↔", etc. op-codes, or via simply exiting the emulator[5]. These binary images can be generated directly using the emulator or can be downloaded from the calculator using the serial cable, SAM-BA, and instructions in the main ***WP 34S*** documentation.

The Assembler/Disassembler tool is run from a command line shell[6]. For example, to disassemble a set of programs saved from flash page 0, the following command may be used:

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

Figure 4.1: Disassembler Example Command-Line

The "-dis" command line "switch" tells the tool to run in disassembler mode. The above command will read the "wp34s-3.dat" binary image[7], detect how many steps are in the one or more programs within the binary file, and print out 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. The output listing, "myProgs.wp34s", might look something like this:

```
// Source file(s): wp34s-3.dat
001 LBL'MOD'
002 RMDR
003 RCL L
004 x[<->]y
005 x<0?
006 +
007 RTN
// 7 total words used.
// 8 total words used.
// 6 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. These asterisks on the labels are also ignored by the assembler.

The tool can also take ASCII source files and turn them into binary flash images as well. In order to assemble a source listing into a binary image for the flash, a command similar to following may be used:

The resultant binary flash image, in this case "wp34s-3.dat", can be loaded directly into either the emulator or the appropriate flash page in the calculator using instructions found in the main ***WP 34S*** calculator documentation.

---

[5] 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.

[6] There currently is no GUI, however, some plans may be underway to create such a version.

[7] 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.

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

Figure 4.3: Assembler Example Command-Line

The Assembler/Disassembler is located in the "`./trunk/tools/`" directory within the SVN development tree.

4.1 Disassembler Instructions

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

As seen in Figure 4.4, to invoke the disassembler, the tool requires the "`-dis`" switch and the name of the binary image to disassemble[8]. If the "`-o`" switch is not given in disassembler mode, the output is sent to the screen (STDOUT). This can be redirected into a file as well, hence both invocations shown below result in 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 with disassembly mode. The "`-s <# of asterisks>`" switch 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 this switch is not present (or if a value of 0 was given), no asterisks will be prepended[9].

```
$ wp34s_asm.pl -dis wp34s-3.dat -s 2 -o myProgs.wp34s
```

Figure 4.5: Disassembler Command-Line For 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:

```
// Source file(s): wp34s-3.dat
001 **LBL'MOD'
002 RMDR
003 RCL L
004 x[<->]y
005 x<0?
006 +
007 RTN
// 7 total instructions used.
// 8 total words used.
// 6 single word instructions.
// 1 double word instructions.
```

Figure 4.6: Source Example with Label Asterisks

[8] It should be noted that the order of the command line switches is not important to the assembler/disassembler. Only those switches that take an extra argument are required to be in sequence.
[9] Asterisks in front of the `LBL` op-codes are ignored by the assembler.

Additionally, the disassembler can be asked to suppress the the step numbers as well. This is sometimes useful when doing text comparisons[10] with previous listings using 'diff-ing' tools[11].

```
$ wp34s_asm.pl -dis wp34s-3.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-3.dat
LBL'MOD'
RMDR
RCL L
x[<->]y
x<0?
+
RTN
// 7 total instructions used.
// 8 total words used.
// 6 single word instructions.
// 1 double word instructions.
```

Figure 4.8: Source Example with No Line Numbers

The disassembler prints a diagnostic statistic set 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.

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

4.2 Assembler Instructions

The assembler takes one or more ASCII source listings and translates them into a single flash 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 506-word limit within any given page of flash memory. Thus, the user must keep the total length of a concatenated set of programs in mind when combining source programs. The statistics printed at the bottom of the disassembly listings can be useful for this. If the 506-word limit is exceeded, the assembler will abort with an appropriate error. In this case, the user must re-balance their assembly source file and/or their mix of assembly source files to ensure that the 506-word limit is not exceeded.

Hint: Programs can access alpha labels in other flash pages and/or RAM from the one they are currently running within. Therefore a program or set of programs can be created that use `XEQ` or `GTO` 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[12]:

After a successful assembly run, the assembler 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.

---

[10] 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.
[11] Examples of differencing tools include '`diff`', '`tkdiff`', '`meld`', etc.
[12] 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
```

```
$ wp34s_asm.pl myProgs.wp34s -o wp34s-3.dat
// CRC16: 520A
// Total words: 32
// Total steps: 31
```

Figure 4.9: Generic Assembler Command-Line

There are a number of options that can be used when assembling a source file as well.

If the user's program does not consume the full 506-word limit within any given flash page, the area beyond the user's program is usually filled with the ***WP 34S*** op-code for "`ERR 03`"[13]. It is possible to program any 16-bit value desired into these unused words by using the "`-f ####`" switch. The numeric value of the "`####`" will be written to these unused words[14].

```
$ wp34s_asm.pl myProgs.wp34s -o wp34s-3.dat -f FFFF
```

Figure 4.10: Assembler Command-Line With Non-Standard Flash Fill

C-style comments can be included in the user's source. Both single line ("`//`") and multi-line comments ("`/* ... */`") 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
```

Figure 4.11: Assembler Source With Comment Styles

The source fragment shown in Figure 4.11 will produce an identical binary image to the ones in the previous examples.

As mentioned in the Disassembler section, though the disassembler writes out program step numbers in its output, these are neither required nor translated during the assembly process[15].

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 13 for more details on how to generate a mnemonic syntax guide table.

```
$ wp34s_asm.pl gc.wp34s fp.wp34s mod.wp34s -o wp34s-3.dat
```

Figure 4.12: Assembling Multiple Source Files to One Image

[13] This instruction was chosen by the ***WP 34S*** design team just in case the calculator mistakenly executed program steps from this unused area. The "`ERR 03`" instruction will cause the program to stop and display this error.

[14] An unprogrammed or empty flash word has the value of 0xFFFF so using the following optional command line switch may result in a longer lived flash memory within the calculator, and a quicker programming time: "`-s FFFF`".

[15] 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 assembler can be used to concatenate several ASCII source files into a single flash image with a single invocation. Additionally, more than one program unit can be within each source file. The only limitation is that the total number of words[16] cannot exceed the 506-word limit. Figure 4.12 shows an example of using the assembler to translate multiple source files into a single flash image. This technique can be useful for maintaining a library of source files and mixing and matching various programs into a flash image as required.

[16] Most ***WP 34S*** instructions occupy one 16-bit word. However there are a few classes of instruction that take 2 words. Therefore, the 506-word limit does not actually equate to the number of program steps the region can contain. It depends on the instruction mix used.

5 ADDITIONAL INFORMATION

5.1 Generating Reference Op-code List

The ***WP 34S*** Assembler/Disassembler script can create a reference file of legal op-codes 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[17] the ***WP 34S*** recognizes.

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[18].

Figure 5.1 shows a fragment of a syntax-helper file[19] (first and last 5 lines).

```
0000  ENTER[^]
0001  CLx
0002  EEX
0003  +/-
0004  .
...
fded  VW[alpha]+[->]D
fdec  VW[alpha]+[->]L
fded  VW[alpha]+[->]I
fdee  VW[alpha]+[->]J
fdef  VW[alpha]+[->]K
```

Figure 5.1: 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[20] within the mnemonic, the user must retain whitespace in their assembler source files. Many special printing characters are "escaped" by the use of square braces ("`[...]`"). These must also be faithfully reproduced[21].

5.2 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 one op-code definition map, and then reassemble it into the new op-code definition map is a useful feature of the ***WP 34S*** Assembler/Disassembler tool.

Using SVN[22] it is possible to generate an op-code definition map for the ***WP 34S*** calculator revision the program was originally running under. This map can be saved and used to disassemble the user's program[23].

---

[17] At time of writing, this expanded syntax table is greater than 445KB in size!

[18] 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 1194, a suggestion is to name the syntax guide file "`syntax_1194.lst`".

[19] This file is from an ancient SVN version and will likely bare no resemblance to the current version. It is shown to illustrate the table format.

[20] "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.

[21] A planned evolution of the assembler/disassembler project is to make this mnemonic format somewhat more "friendly". However, until then, it is very strict in its mnemonic format.

[22] It is expected that the user can look up SVN usage via the usual Internet resources so only very bare commands will be presented here – without much explanation.

```
$ cd <location of svn working directory>
$ svn up -r <REV #>
$ cd ./trunk
$ make
$ ./Linux/calc opcodes > svn_XXXX.op
$ wp34s_asm.pl -dis wp34s-0.dat -opcode svn_XXXX.op > mySrc.wp34s
```

Figure 5.2: Linux Command-Line to Disassemble from a Specified Op-Code Map

Repeat the last step for as many flash images as are required, placing each output listing in a separate "`*.wp34s`" file.

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

```
$ cd <location of svn working directory>
$ svn up
$ # Observe the SVN number reported at the end of the update.
$ cd ./trunk
$ make
$ ./Linux/calc opcodes > svn_YYYY.op
$ wp34s_asm.pl mySrc.wp34s -opcode svn_XXXX.op -o wp34s-3.dat
```

Figure 5.3: Linux Command-Line to Assemble from a Specified Op-Code Map

Repeat the last step for each flash page required to be translated.

The calculator can now be re-flashed with the latest "`./trunk/realbuild/calc.bin`" 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[24]. 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.

5.3 Automatically Locating Op-code Map

As mentioned above, the script can use either the internal op-code translation map or an external one. The external 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 svn_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 falls back on its internal op-code map table.

Be aware that the `wp34s.op` file is automatically updated by the design team's SVN build process whereas the assembler script (and its internal table) is not. Hence, the SVN `wp34s.op` file may be more up-to-date than the assembler's internal table with respect to the op-code evolution.

[23] 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.
[24] 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).

6 EXAMPLES LIBRARY

The "`./library/`" directory distributed with the ***WP 34S*** project includes a number of ***WP 34S*** programs either culled from the calculator's internal XROM database or contributed by users. These programs are intended 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 source files for the ***WP 34S***. A README_ASM file is included in the directory with more information.

7 SOURCE OF PERL PACKAGES

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

7.1 Linux

A Perl interpreter is included with virtually every Linux distribution available[25]. With Linux, there is usually nothing more that needs to be done beyond running the examples as shown in the preceding sections.

7.2 Windows

Perl must be added separately to a Windows system. Fortunately there are several very good Perl packages available for Windows[26].

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

7.2.1 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"[27]. If you are well versed in Linux environments, cygwin will be immediately familiar.

7.2.2 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 7.1 shows how this is done:

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

Figure 7.1: Running Script Under Strawberry Perl

The Strawberry Perl installer does not setup the file associations for Perl scripts (as ActiveState does). There are many good reference for "correcting" this oversight[28].

7.2.3 ActiveState Perl

ActiveState 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.

ActiveState 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

[25] Actually, the author is unaware of any that does *not* include a Perl package.
[26] And some very poor packages as well (*caveat utilitor*!).
[27] 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.
[28] See http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/ftype.mspx for an example of how this may be done.

automatically recognize Perl scripts and launch the interpreter accordingly. Hence, other than the 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 7.2: Running Script Under ActiveState Perl

7.2.4 Native Windows Executable

There is an experimental Windows stand-alone EXE file built using a Perl-to-executable module called PAR::Packer[29]. PAR::Packer includes a tool called "pp" which "packages" the Perl script along with Perl libraries into an executable form[30].

This assembler/disassembler 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 7.3: Running the Windows Executable

The executable should be usable on virtually any Windows installation with no additional libraries or files required[31]. 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!).

7.3 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.

[29] Available from CPAN. Current version v1.009.
[30] 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.
[31] The executable version of the script will also automatically locate the `wp34s.op` file, if available, as described in Section 5.3.

8 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 with a single command-line invocation.

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`.
- Any instruction capable of using either a numeric or single letter alpha `LBL` as a target can now use a 'soft' symbolic label instead. This allows for opportunistic `LBL` re-use and automatic `LBL` insertion, where required. 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 available below.

8.1 Running the Preprocessor

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 directly.

8.1.1 Stand-Alone Preprocessor

Figure 8.1 shows the ***WP 34S*** PP being run directly from the command line in stand-alone mode[32].

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

Figure 8.1: 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 top portion of this document.

8.1.2 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 8.2, the '-pp' switch tells the ***WP 34S A***ssembler to automatically feed any source files through the ***WP 34S*** PP tool prior to assembling them. By default, the intermediate source code is

[32] 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 8.2: Running WP 34S PP Through the WP 34S Assembler

written to a file called `wp34s_pp.lst`. Even though it is called a `*.lst` file, this file is in the normal `*.wp34s` 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.

8.2 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 (99), the `BACK` and/or `SKIP` instruction can no longer be used. In this case the user must insert a `LBL` at the target location and use a `GTO` to branch to that target step.

By replacing occurrences of `BACK`, `SKIP`, and `GTO` 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 8.3 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.

```
001 **LBL'8Qu'
002 CLREG
003 8
004 STO 11
005 RCL 11
006 x=? 00
007 SKIP 22
008 INC 00
009 STO[->]00
010 INC 10
011 RCL 00
012 STO 09
013 DEC 09
014 RCL 09
015 x=0?
016 BACK 11
017 RCL[->]00
018 RCL-[->]09
019 x=0?
020 SKIP 05
021 ABS
022 RCL 00
023 RCL- 09
024 x[!=]? Y
025 BACK 12
026 DSZ[->]00
027 BACK 17
028 DSZ 00
029 BACK 03
030 RCL 10
031 RTN
```

Figure 8.3: 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 must count the intervening steps to the desired target and insert that value into the instruction (remembering to offset the number by one if using a `SKIP`!). If the code is ever modified to insert or delete steps between the branch instruction and the target step, all affected branches must be revisited and repaired. Furthermore,

if the program is later modified to require the branch to be greater than 99 steps, the source must be modified to replace these out-of-scope branches with GTO statements, and LBLs must be inserted at the targets. Note that the insertion of these new LBLs may drive additional SKIP/BACK branches out-of-scope as well, 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 analyses 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 99, the JMP will be replaced by a BACK instruction. If the offset is positive and less than 99, the JMP will be replaced with a SKIP instruction. If the offset is greater than 99, regardless of the direction, the JMP will be replaced by a GTO with a matching numeric LBL inserted[33] 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 8.2.1 for more details.)

```
001          *LBL'8Qu' // Entry point
002          CLREG
003          8
004          STO 11
005 loop::   RCL 11
006          x=? 00
007          JMP done // SKIP 22
008          INC 00
009          STO[->]00
010 again::  INC 10
011          RCL 00
012          STO 09
013 loop2::  DEC 09
014          RCL 09
015          x=0?
016          JMP loop // BACK 11
017          RCL[->]00
018          RCL-[->]09
019          x=0?
020          JMP Not0 // SKIP 05
021          ABS
022          RCL 00
023          RCL- 09
024          x[!=]? Y
025          JMP loop2 // BACK 12
026 Not0::   DSZ[->]00
027          JMP again // BACK 17
028          DSZ 00
029          BACK 03
030 done::   RCL 10
031          RTN
```

Figure 8.4: 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 8.4 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

[33] 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 – only local types.

symbolic labels[34] 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 JMPs).

The output of ***WP 34S*** PP is shown in Figure 8.5. 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 future reference[35]. Since this program was so short, none of the JMP pseudo instructions were required to be 'promoted' to GTO. See Section 8.2.1 for an example of JMP→GTO promotion.

```
001 /*         */ *LBL'8Qu'
002 /*         */ CLREG
003 /*         */ 8
004 /*         */ STO 11
005 /* loop::  */ RCL 11
006 /*         */ x=? 00
007 /*         */ SKIP 22 // JMP done
008 /*         */ INC 00
009 /*         */ STO[->]00
010 /* again:: */ INC 10
011 /*         */ RCL 00
012 /*         */ STO 09
013 /* loop2:: */ DEC 09
014 /*         */ RCL 09
015 /*         */ x=0?
016 /*         */ BACK 11 // JMP loop
017 /*         */ RCL[->]00
018 /*         */ RCL-[->]09
019 /*         */ x=0?
020 /*         */ SKIP 05 // JMP Not0
021 /*         */ ABS
022 /*         */ RCL 00
023 /*         */ RCL- 09
024 /*         */ x[!=]? Y
025 /*         */ BACK 12 // JMP loop2
026 /* Not0::  */ DSZ[->]00
027 /*         */ BACK 17 // JMP again
028 /*         */ DSZ 00
029 /*         */ BACK 03
030 /* done::  */ RCL 10
031 /*         */ RTN
```

Figure 8.5: 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[36]. 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.)

---

[34] 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.

[35] 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. There is a push to have the tool upgraded to preserve initial comments (if I can figure out how to do it!).

[36] For the moment, 'hard' BACK and SKIP instructions will be left untouched. A future version may detect these cases and convert them to 'soft' JMPs 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 the is such a thing as legacy code for a device that in barely old enough to be out of diapers!).

8.2.1 JMP Target Offset Greater Than Maximum Offset Allowed

If the branch offset is greater than the maximum allowed (usually 99), the JMP pseudo instruction is replaced by a GTO 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 100 steps), the next example shows this mechanism at work but with the maximum allowed offset artificially lowered[37] 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 GTO/LBL pairs. Observe that the JMP Not0 was converted to SKIP 05 and not to a GTO 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'
/*         */ CLREG
/*         */ 8
/*         */ STO 11

/* loop::  */ RCL 11
/*         */ x=? 00
/*         */ SKIP 22 // JMP done
/*         */ INC 00
/*         */ STO[->]00

/* again:: */ INC 10
/*         */ RCL 00
/*         */ STO 09

/* loop2:: */ DEC 09
/*         */ RCL 09
/*         */ x=0?
/*         */ BACK 11 // JMP loop
/*         */ RCL[->]00
/*         */ RCL-[->]09
/*         */ x=0?
/*         */ SKIP 05 // JMP Not0
/*         */ ABS
/*         */ RCL 00
/*         */ RCL- 09
/*         */ x[!=]? Y
/*         */ BACK 12 // JMP loop2
/* Not0::  */ DSZ[->]00
/*         */ BACK 17 // JMP again
/*         */ DSZ 00
/*         */ BACK 03

/* done::  */ RCL 10
/*         */ RTN
```

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

Figure 8.6: 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!

8.3 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, GTO, etc[38]. If no 'hard' LBL exists at the target location identified by the matching 'soft' label, ***WP 34S*** PP will

[37] 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 99-step default and sets the maximum allowed offset to 90 instead. The reasons for this are beyond the scope of this document.

[38] Besides LBL, at time of writing, the instructions that can take a 'local' label include: LBL?, XEQ, GTO, [SIGMA], [PI], SLV, f'(x), f"(x), and [integral].

automatically inject a unique numeric-type LBL instruction for this purpose[39]. 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[40].

The example in Figure 8.7 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
        [cmplx]STO L
        [cmplx]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
```

Figure 8.7: LBL-less Example – WP 34S PP Source

As shown in Figure 8.8, ***WP 34S*** PP detects this relationship and automatically inserts a LBL 99 instruction at the appropriate location[41]. Additionally, the 99 was also substituted in the XEQ instruction as well (with the original source instruction moved into a comment field).

[39] ***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.
[40] Equivalent LBLs are defined as more than one consecutive LBL.
[41] 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. Note that it is possible to exhaust the numeric LBL space but with 506 words available in every flash/RAM slot, this would require an average LBL density exceeding one per every 5 instructions – somewhat unlikely. However, should this happen, ***WP 34S*** PP will issue an appropriate error message and exit. Additionally, the astute reader will have observed that the LBL is injected one step before the symbolically labeled instruction. This is intention and aids in possibly (re)using the symbolically labeled instruction with another SKIP instruction generated by the JMP pseudo instruction. (Moving forward, if a SKIP lands on a LBL it 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.)

```
001 /* Vabs:: */ LBL'VAB'
002 /*        */ XEQ 99 // XEQ Vsub
003 /*        */ x[<->] T
004 /*        */ STO J
005 /*        */ DROP
006 /*        */ [cmplx]STO L
007 /*        */ [cmplx]DROP
008 /*        */ RTN
009 /*        */ LBL 99
010 /* Vsub:: */ ENTER[^]
011 /*        */ x[^2]
012 /*        */ RCL Z
013 /*        */ RCL[times] T
014 /*        */ +
015 /*        */ RCL T
016 /*        */ RCL[times] T
017 /*        */ +
018 /*        */ [sqrt]
019 /*        */ RTN
```

Figure 8.8: LBL-less Example – WP 34S PP Output

8.4 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 8.9.

A recent development has seen the addition of triple alpha strings using the quoted alpha mnemonic. Shown in the bottom portion of Figure 8.9.

```
// 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 8.9: 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[42].

As a convienience, ***WP 34S*** PP will also recognize embedded spaces and correctly translate these to the correct '[space]' mnemonic as well. See Figure 8.10 for an example.

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

Figure 8.10: Double Quoted Alpha String Example

[42] 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 3rd character nulled out (4 bytes).

All three program fragments will result in exactly the same text in the ***WP 34S*** alpha display. In the case of Figure 8.10, 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 8.9.

```
001 CL[alpha]
002 [alpha]'WP[space]'
003 [alpha]'34S'
004 [alpha]VIEW
005 PSE 08
// 5 total instructions used.
// 7 total words used.
// 3 single word instructions.
// 2 double word instructions.
```

Figure 8.11: Doubled Quoted Alpha String Disassembly

9 HELP

Both the ***WP 34S*** Assembler/Disassembler and Preprocessor (PP) 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_p.pl -h
```

Figure 9.1: Invoking the HELP Screens