PP "C" Preprocessor User Guide

PP Version 91.1
6/1/91

Program by Gary Oliver & Kirk Bailey
Documentation by Logical Systems

Contents

Introduction

Overview

PP is a preprocessor for the "C" programming language. It conforms to the ANSI "C" standard (ANSI X3.159-1989), and contains options for backward compatibility with programs written for non-ANSI preprocessors. Although PP is designed for the "C" language, its facilities can be used to aid the programming process in other languages (it is particularly useful for augmenting assemblers which lack file inclusion and macro capability). PP contains a number of novel features implemented via ANSI suggested extension techniques which greatly enhance its usefulness with "C" and other languages.

PP is written in the "C" programming language and can be (relatively), easily configured to run on a wide range of machines. PP currently runs on various platforms running BSD and SYS5 UNIX and under MS-DOS using Microsoft "C" version 6 or Borland "C" version 2.0. PP may also be run under MS-DOS with the QCX Z80, and TCX Transputer, cross compilers. Finally, last (and least), it runs under Z80 CP/M 2.2 using the QC version 3.2 and 4.0 compilers. An experimental version is also running on the VAX under VMS.

Unlike the rest of this software development package, PP has been placed in the public domain. This means no copyright is claimed and you may do what you wish with it. As a point of courtesy, please retain the authorship information in the source code and documentation.

If you are familiar with how a "C" preprocessor works skip the rest of this section:

The basic function of the preprocessor is reading "C" source code files and handling the file inclusion, conditional compilation, macros and other lexical manipulations which are requested by the programmer. When finished, the preprocessor writes out a simplified version of the input source code file out for later compilation by the "C" compiler proper. The original "C" compilers, produced by Bell Labs and others, all organized the preprocessor activity (often called a "pass" in compiler parlance), to operate in this fashion. As computer systems with larger memories become available it is getting common to see "C" compilers which integrate the preprocessor pass with the rest of the compiler as one program. In any event, the preprocessor can be thought of as a distinct entity and in the implementation of PP this is literally the case.

System Requirements

PP was originally written using the QC "C" compiler (version 3.2), under CP/M 2.2 on the Z80. It is run in other environments by recompiling and changing some configuration parameters (or worst case by modifying the source code for compilers with non-standard libraries or environments). Configurations for which it has been tested include running under MS-DOS for use with the QCX Z80, and TCX Transputer, cross compilers and under various flavors of UNIX. PP is modest in its requirements for memory and disk space and should have little trouble running in a environment which currently supports any sort of "C" compiler.

Usage

The general form of the PP command line is:

```
pp <input_filename> -[options]*
```

The basic idea is to specify the required input filename (complete with filename extension), followed by any options needed. Note that in this case, since no explicit output filename has been specified, the output filename will default to that of the input filename, but with an extension of ".pp" in place of any extension the input filename had.

Examples

Assume you wish to preprocess a "C" program stored on file "foo.c"; The syntax needed is simply:

```
pp foo.c
```

No sweat right! If you wanted to write the output to some file other than "foo.pp", say "foobar", you would use the "-o" option flag followed by the desired output filename:

```
pp foo.c -o foobar
```

Option Information

As seen above with the "-o" option, an option flag may need a following parameter, although many option flags are simple switches which may be grouped together following a common "-" option flag lead-in. For instance, if you wish to see a debugging statistics summary when PP exits AND you do not wish to abort on errors (return a non-zero value for the PP exit code), the following command line will do it (turning "foo.c" into "foo.pp"):

```
pp foo.c -se
```

Where the "s" indicates you want the debugging statistics summary and the "e" inhibits PP from returning an error code. The above result could also be obtained by separating the option flags:

```
pp foo.c -s -e
```

Please note, although in the above examples the option flags were in lowercase, uppercase is also allowed (some systems support nothing else)!

Option Descriptions

The following descriptions detail all the option flags available with PP, what each does, and what additional parameters are required. If you aren't generally familiar with the "C" preprocessor you should consult one of the many available books on "C" programming. Now, on to the command line options:

`-c 0`

If this option flag is given the preprocessor will examine the contents of strings when looking for formal parameter names in the replacement list for a macro. This option is disabled by default (in conformance with ANSI desires), and can have unexpected side effects when mixed with the ANSI "#" and "##" operators (or strings which contain "#" or "##" character sequences). Another side effect is that any "whitespace" inside strings will be collapsed to a single space. This option is provided for use with "C" programs which need this capability but new use of it is heavily discouraged. The same effect may be had in a more selective fashion by using the "#pragma arg_string" and "#pragma no arg_string" directives presented in the section below labeled **PP Predefined Symbols**. The best alternative is to use the ANSI "#" operator together with the string concatenation facility of ANSI style "C" compilers (including TCX). Whitespace is allowed between the "c" and the "0" on the command line.

`-c 1`

This option flag instructs PP to allow macro replacement to occur inside "#pragma asm" blocks. This option is disabled by default. See the **PP Pragma Support** section for a description of what "#pragma asm" does. Whitespace is allowed between the "c" and the "1".

`-c 2`

This option flag allows PP to correctly handle "recursive" comments. These comments are disallowed by ANSI (the option is disabled by default), but are occasionally useful. As a general rule, portability dictates that you avoid using this facility. However, as a temporary tool during program development, the ability to "comment out" sections of code is handy. Whitespace is allowed between the "c" and the "2".

`-c 3`

This option flag instructs PP to rescan the fully argument replaced (and argument macro expanded), result of a macro call for further preprocessing directives in conjunction with the final re-scan of the macro replacement list for additional macros to expand. If directives are encountered they are executed in the normal fashion. This option is disabled by default and is expressly prohibited by ANSI. It is included in PP since it makes multi-line macros much more useful (they of course, are also not included in ANSI). Regardless of the setting of this option, "#pragma" directives are ALWAYS recognized during the re-scanning of the macro replacement string. Whitespace is allowed between the "c" and the "3". A few points about these "nested" directives are in order:

1. Any use of "##" or "#" on a nested "#define" or #pragma define" will not give the expected result since the "##" and "#" operators are evaluated and discarded BEFORE the subsequent re-scan of the body runs into the "nested" directive which contained them.

2. Any line in the replacement list which begins with "#" is not automatically macro replaced UNLESS the arguments to that directive would be normally macro replaced anyway ("#define" and "#undef" being among those which would not). Note that the directive name is never macro replaced (unless it was originally the result of formal parameter substitution and macro replacement). One consequence of this is that ANSI macro "scoping" rules for macro formal arguments may be violated since the directive forms a new "top-level" basis for evaluating formal parameter scope. The normal scoping implied by the "nesting" still holds however (ie. if macro A is called recursively or co-recursively within the replacement list for macro A). The distinction between "body" scoping and "formal parameter" scoping here is subtle, read (and re-read), the ANSI specification to get a better idea of how ANSI has changed the rules for macro substitution.

3. The syntax for these directives doesn't conflict with ANSI since ANSI disallows a "#" which doesn't preceed a formal parameter. By definition, if the "#" qualified as an ANSI operator then the "#" would end up disappearing and a "stringized" formal parameter would replace it (which doesn't have the same form as a directive).

`-c 4`

This option flag implements the "stacking" model for "#define" and "#undef" directives. In this model each subsequent "define" of the same symbol "hides" the previous definition of the symbol (which can be later re-enabled by a "#undef"). ANSI mandates a "binary" model for "#define/#undef" and thus this option is disabled by default. The ANSI model is "safer" programming practice and its use is encouraged. Whitespace is allowed between the "c" and the "4".

```
-c 5
```

This option forces PP to recognize and translate incoming Trigraph character sequences. Although ANSI mandates this, the option is disabled by default since it can "break" existing code and is of somewhat limited utility (depends on what your national character set is of course). Whitespace is allowed between the "c" and the "5".

```
-d <macro_symbol_name>[=<macro_expression>]
```

The "-d" option flag allows you to define a preprocessor macro symbol from the command line. If you do not specify the "=<macro_expression>" clause the symbol is given a default value of "1". Note that if the expression clause IS present, it must contain no whitespace between the symbol name and the equals sign or between the equals sign and the expression (or within the expression). The following examples define the preprocessor symbol "hi" to a value of "1" within file "foo.c":

`pp foo.c -dhi` Or equivalently: `pp foo.c -dhi=1`

```
-e
```

As mentioned previously, this option flag tells PP not to return an abort exit return value (a non-zero value), even if errors were detected.

```
-i <include_file_pathname>
```

This option flag lets you specify a non-standard pathname for include files to be found on. Only one include file path should be specified for each "-i" option flag. The basic search strategy for include files for non-CP/M versions of PP is as follows:

If the #include filename is bracketed by quotes, search the current directory first to find the specified include file. If PP can't find the file, it searches on the directories given by the "-i" switches in the order they appear on the command line. If PP still can't find the file it then checks to see if an environment variable by the name of "PPINC" exists, if so it interprets the associated value as a ';' separated list of directories to search. If the PPINC variable doesn't exist PP will use the default system configured include file pathname for the "final" search path ("/usr/include" or "/include" depending on how the system is configured, see the **PP Default Include File Pathname** section for a description of DFLT_PATH). If the #include filename was bracketed by angle brackets instead of quotes, PP omits the initial search of the current directory.

If you have a CP/M version of the preprocessor, things get a little more complicated. First of all, since CP/M doesn't support the concept of a tree structured file system, directories don't really exist. To compensate for this a pair consisting of a CP/M drive and user number is used in place of a directory pathname. Thus, if you wished to specify a include file search on drive C, user number 5 you would use (with our old standby "foo.c"):

```
pp foo.c -i c5
```

Note that the system configured include file pathname is "A0" (which is searched last). All the stuff so far operates in an analogous fashion to the non-CP/M case, but here is where things get different: If a file by the name of "path.ppi" is present on the system configured include file pathname, it is read and the contents used as if they were read from a series of "-i" switches on the command line. If this file is present its contents OVERRIDE the system configured include file pathname. Thus the basic strategy for searching for CP/M include files is as follows:

If the #include filename is bracketed by quotes, search the current drive and user number for the include file. If PP can't find it there it searches the "-i" drive/user-number pairs that were on the command line in the order they were present. If it hasn't yet found the include file it searches the drive/user-number pairs it found on file "path.ppi" (if the file existed on A0, if not search A0 for the include file). If the include filename was enclosed by angle brackets instead, the above process occurs, but the initial search of the current drive and user number is omitted.

```
-l a
```

If this option flag is given, the output of the preprocessor contains the abbreviated form of line number information via "#<line-number>" pseudo-preprocessor directives. Whitespace is allowed between the "l" and the "a".

```
-l l
```

If this option flag is given the output of the preprocessor contains the expanded form of line number information via "#line <line-number>" pseudo-preprocessor directives. Whitespace is allowed between the two "l"s. This option flag is the default if no "-l" option flag is given.

```
-l n
```

If this option flag is given, the output of the preprocessor doesn't contain either of the two forms of line number information. Note: if this option is used there is no way to tell where a particular output line written by the preprocessor was originally read from. This causes error messages written by any downline program (such as the compiler), to contain incorrect filename and line number information.

`-o <output_filename>`

As mentioned, this option flag allows you to explicitly specify the output filename for PP to use.

`-s`

This option flag allows you to generate a statistics summary when PP terminates. This option flag is only available when PP is compiled with the DEBUG configuration definition set to TRUE in "pp.h" (the default in the release versions of PP). This option provides information of interest to those debugging PP or bringing it up on systems with marginal resources (particularly memory).

`-t a <string_of_characters>`

This option flag causes all the characters present in the "string_of_ characters" to be added to the preprocessors's idea of what a alpha character is (a letter). This affects token recognization and all other cases where the preprocessor requires a alpha to be present.

`-t r <string_of_characters>`

Like "-t a" above, but this option flag removes the corresponding characters from the alpha class.

`-u <macro_symbol_name>`

This option flag allows you to do the equivalent of a #undef on one of the predefined preprocessor symbols (see the **PP Predefined Symbols** section for information about what initial symbols PP defines depending on what system it is configured for).

`-v`

This option flag toggles PP between the verbose and quiet output modes. The default output mode is dependent on the setting of the VERBOSE comfiguration option in "pp.h" which in turn is dependent on the host operating system with which PP is being used. On the Macintosh, PP defaults to the quiet output mode, on other systems the default is verbose. See Appendix B for more information on the VERBOSE configuration option. Assuming the verbose mode is ON, PP will print out identification information followed by a record of what it does. This record includes the output file name, which input files were processed, how many preprocessor symbols remained defined when PP finished and how many symbols were used (worst case), during processing.

`-z`

This option flag turns on the debugging messages that PP can print during operation. This option is only available if the DEBUG definition in "pp.h" is set TRUE when PP is compiled (which is the case for the release versions). The debugging output is somewhat cryptic, but since you probably don't need it unless you change PP, you should understand the source code well enough to decipher it.

`-?`

This option flag causes a mini version of this option description section to be printed out to you. PP will not do anything else when this happens, including preprocessing a file! You get the same effect as this option by making any sort of command line syntax error.

PP Syntax and Semantics

PP conforms to the syntax and semantics of a ANSI "C" Standard preprocessor. As mentioned, it also contains compatibility options for use with programs written using previous flavors of "C" preprocessors. For most of the common questions you should consult one of the many "C" programming books. A good one is:

"C" A Reference Manual
Samuel P. Harbison/Guy L. Steele Jr.
Prentice-Hall, Inc.
Englewood Cliffs, NJ 07632, USA

This book covers both traditional and ANSI style preprocessors and is a reasonable reference. For details about ANSI "C" macro substitution rules the only known reference with enough meat is the Draft Standard itself and only then in conjunction with the examples it provides (the text is ambiguous about some of the finer points).

One part of the standard which is implementation dependent is the selection of which #pragma statements you support.

PP Pragma Support

PP explicitly supports 27 #pragma directives. They are:

```
#pragma   arg_string
```

Using this pragma is equivalent to setting the "-c0" command line configuration switch (ie. it allows macro formal parameters to be recognized within string constants).

```
#pragma   asm
```

This is used only with either the QC, QCX or TCX compilers. It causes all the code between this and the next "#pragma endasm" to be output to the compiler without changes as assembly language inline code (a opening "#asm" is emitted to let the compiler know what's going on). Note that whether macro substitution occurs in the code between the pragmas is dependent on the setting of the "#pragma [no] asm_expand" pragma (see below), or the "-c1" command line option flag. "#pragma asm" statements may not be nested.

`#pragma asm_expand`

This pragma is used with "#pragma asm"/"#pragma endasm" directives to enable macro expansion within inline assembly language blocks.

`#pragma   comment_recurse`

Using this pragma is equivalent to setting the "-c2" command line configuration switch (ie. PP allows recursive comments).

`#pragma   define`

Identical to a normal "#define" except since it's a "pragma" it will always be recognized and processed in macro replacement lists unlike the non-pragma variant which is dependent on the setting of a configuration option. The configuration option for the non-pragma variant is controlled by the "-c3" command line option flag and the "pragma [no] macro_rescan" pragmas.

`#pragma   elif`

Identical to a normal "#elif" except since it's a "pragma" it will always be recognized and processed in macro replacement lists unlike the non-pragma variant which is dependent on the setting of a configuration option. The configuration option for the non-pragma variant is controlled by the "-c3" command line option flag and the "pragma [no] macro_rescan" pragmas.

`#pragma   else`

Identical to a normal "#else" except since it's a "pragma" it will always be recognized and processed in macro replacement lists unlike the non-pragma variant which is dependent on the setting of a configuration option. The configuration option for the non-pragma variant is controlled by the "-c3" command line option flag and the "pragma [no] macro_rescan" pragmas.

`#pragma   endasm`

The other half of the "#pragma asm" directive above. This directive emits a "#endasm" directive to the compiler and terminates the inline assembly language inclusion. Macro substitution is unconditionally enabled by this directive (since it may have been turned off).

```
#pragma   endif
```

Identical to a normal "#endif" except since it's a "pragma" it will always be recognized and processed in macro replacement lists unlike the non-pragma variant which is dependent on the setting of a configuration option. The configuration option for the non-pragma variant is controlled by the "-c3" command line option flag and the "pragma [no] macro_rescan" pragmas.

```
#pragma endmacro
```

Terminates the most recent un-terminated "#pragma macro name(...)" directive.

```
#pragma error message_text
```

Like "#pragma message", but the occurence of this directive is counted as an error by PP and affects the exit return value (unless a "-e" option overrides it). The new draft ANSI "C" standard includes a "#error" directive which duplicates this pragma (except the string to be displayed is contained in quotes), thus the use of this pragma for new work is discouraged.

```
#pragma   if
```

Identical to a normal "#if" except since it's a "pragma" it will always be recognized and processed in macro replacement lists unlike the non-pragma variant which is dependent on the setting of a configuration option. The configuration option for the non-pragma variant is controlled by the "-c3" command line option flag and the "pragma [no] macro_rescan" pragmas.

```
#pragma   ifdef
```

Identical to a normal "#ifdef" except since it's a "pragma" it will always be recognized and processed in macro replacement lists unlike the non-pragma variant which is dependent on the setting of a configuration option. The configuration option for the non-pragma variant is controlled by the "-c3" command line option flag and the "pragma [no] macro_rescan" pragmas.

```
#pragma   ifndef
```

Identical to a normal "#ifndef" except since it's a "pragma" it will always be recognized and processed in macro replacement lists (unlike the non-pragma variant which is dependent on the setting of a configuration option). The configuration option for the non-pragma variant is controlled by the "-c3" command line option flag and the "pragma [no] macro_rescan" pragmas.

```
#pragma macro name(parameters...)
```

This pragma allows multiple line macros to be defined. The macro operation is the same as the normal "#define", but the body of the macro extends to the next matching "#pragma endmacro". Note that this type of macro definition can be nested! You may also include other "#pragma" directives in the replacement list (or normal directives if the "-c3" command line option has been given or the equivalent via a "pragma macro_rescan" directive).

```
#pragma   macro_rescan
```

Using this pragma is equivalent to setting the "-c3" command line configuration switch (ie. it allows non-pragma directives to be recognized in macro replacement lists). See the description in the **Option Descriptions** section (option "-c3"), for more information about the effects of this pragma.

```
#pragma   macro_stack
```

Using this pragma is equivalent to setting the "-c4" command line configuration switch (ie. the "stacking" model of "#define/#undef" directives is used).

```
#pragma message message_text
```

Prints everything on the line after the "message" keyword to standard output. Any macros in "message_text" are expanded.

```
#pragma no arg_string
```

Like "#pragma arg_string", except macro formal parameter recognition within string constants is DISABLED (per ANSI specification).

```
#pragma no asm_expand
```

Like "#pragma asm_expand" above, but this disables macro expansion within inline assembly language blocks. This condition is the default if no "#pragma asm_expand"/"#pragma no asm_expand" directives are encountered.

```
#pragma   no comment_recurse
```

Like "#pragma comment_recurse", except recursive comments are DISABLED per ANSI specification.

```
#pragma no macro_rescan
```

Like "#pragma macro_rescan", except recognition of non-pragma directives in macro replacement lists is DISABLED per ANSI specification.

```
#pragma no macro_stack
```

Like "#pragma macro_stack", except the ANSI "binary" model of "#define/#undef" is used instead of the stacking model.

```
#pragma   no trigraph
```

Like "#pragma trigraph", except Trigraph sequence recognition and translation is DISABLED.

```
#pragma   trigraph
```

Using this pragma is equivalent to setting the "-c5" command line configuration switch (ie. PP will recognize and translate Trigraph sequences on input per ANSI specification).

```
#pragma   undef
```

Identical to a normal "#undef" except since it's a "pragma" it will always be recognized and processed in macro replacement lists; unlike the non-pragma variant which is dependent on the setting of a configuration option. The configuration option for the non-pragma variant is controlled by the "-c3" command line option flag and the "pragma [no] macro_rescan" pragmas.

```
#pragma   value
```

This pragma allows a simple form of preprocessing time arithmetic. The arguments to it are identical to a "#if" (or "#pragma if"), but the effect is that the evaluated value is converted to a decimal number and written to the output stream.

Note that any "#pragma" directives which are not recognized by PP are passed through to the output file with the understanding that somewhere down the line another program WILL recognize and process them.

PP Predefined Symbols

PP supports several predefined macro symbols. The following are defined for all host and target system configurations:

```
__LINE__
```

Is replaced by the line number of the input file on which it occurred.

```
__FILE__
```

Is replaced by the filename of the current input file, bracketed with double quotes.

The above two symbols allow you to create completely customized error checking and reporting. One good use is the creation of an "ASSERT" macro which checks assertions about "C" programs. For example:

```
#define   ASSERT(test) if(! (test)) \
     printf("Assertion FALSE on file: %s, line:%d\n" \
          ,__FILE__,__LINE__); \ else;
```

You can then use this macro to "alert" you when the various things you took for granted in writing the program no longer hold (particularly portability dependencies). For instance, suppose you are depending on the fact that of two configuration macro definitions, only one of them is set TRUE at any one time. Let FALSE be defined to be 0, TRUE be defined to be 1, and the two configuration options be CONFIGA and CONFIGB (both set to TRUE to illustrate the technique):

```
ASSERT((CONFIGA + CONFIGB) == TRUE)
```

The above test will cause the ASSERT macro to generate code that will cause your program to let you know that something has been configured incorrectly! Alternately, you could have the body of the ASSERT macro issue a "#pragma error" or "#pragma message" if the conditions you are checking can be detected at preprocessing time (as in the example above).

`__DATE__`

This symbol is replaced by the system local date as of the start of PP execution. The date is in month, day and year format, surrounded by double quotes. For example:

```
"Feb 12 1986"
```

`__TIME__`

This symbol is replaced by the system local time as of the start of PP execution. The time is in 24 hour format with hours, minutes and seconds, separated by colons, and surrounded by double quotes. For example:

```
"23:45:15"
```

The use of the `__DATE__` and `__TIME__` symbols is dependent on a operating system which supports date and time.

`__NEXT__`
`__NOW__`
`__PREV__`

These three symbols are used to control the activity of the unique number generator. The `__NOW__` symbol is replaced by the current value of the generator (an unsigned number which is initialized to zero). Each time PP encounters `__NEXT__` it increments the unique number generator by one and returns the new value. Each time PP encounters `__PREV__` it decrements the unique number generator by one and returns the previous value. These three symbols are designed for use within macro bodies when creation of "local" variable names is useful. The `"##"` operator may be used to glue together the numbers generated this way with other preprocessor tokens. For example:

```
#define   glue(a,b) a##b

#pragma macro ...
__NEXT__
#pragma define label(a,b)     glue(a,b)
#pragma define label1         label(L1,__NOW__)
#pragma define label2         label(L2,__NOW__)
   .....
#pragma endmacro
```

The first macro definition creates a macro which glues its arguments together without macro expanding either of them (because of the semantics of the "##" operator). Then, within the macro where "local" variable names are desired, the call of __NEXT__ generates a new unique number (note that the number is written out by PP, and must be turned into a no-op for whatever language PP is being used with). Following this, the "label" macro serves to provide a layer of insulation between the desired operation (performed by the "glue" macro), and the actual tokens to glue together. This is done since then the formal parameter strings are macro expanded BEFORE being glued together thus allowing the "__NOW__" macro to expand to the current value of the unique number generator. Finally, the last two macro definitions create shortform names for the variables. The variables themselves will be symbols of the form "L1XXX" and "L2XXX" where the "XXX" parts are identical variable length decimal digit strings (since __NOW__ was used instead of __NEXT__).

Note that since "C" preprocessor macro expansion holds off the binding until the very end. If __NEXT__ had been used in either definition it would have generated a different symbol each time "label1" or "label2" is referenced, which is not very useful if you plan on using the symbol more than once!

The unique number generator can also be used for other purposes. It's handy whenever a series of numbers is required at compile time. The __NEXT__, __NOW__ and __PREV__ symbols are not part of the ANSI Draft Proposed "C" Standard and shouldn't be used if portability is paramount. On the other hand, they are VERY useful within multi-line assembly language macros! Since the ANSI standard doesn't allow those either you aren't making your portability problems any worse...

The following predefined macros are used to get the current value of some of the PP configuration options. The value will be "1" if the associated configuration option is enabled and "0" if the option is disabled (they all are disabled by default). For descriptions of the meanings of these see the previous section for the corresponding "#pragma <name>" directive.

```
---------------------------------------------------------------------------------

__ARG_STRING__
__ASM_EXPAND__
__COMMENT_RECURSE__
__MACRO_RESCAN__
__MACRO_STACK__
__TRIGRAPH__
```

The other predefined macro symbols which exist are dependent on which TARGET system PP is generating code for. The possible choices are:

1. **BSD 4.X UNIX systems** (configuration option TARGET set to T_BSD in "pp.h"). The symbols "unix" and "BSD" are both predefined to a value of "1".

2. **Mac's running MPW** (configuration option TARGET set to T_MPW in "pp.h").

3. **Z80 code to run under CP/M, compiled under CP/M** (written using the QC native compiler), configuration option TARGET set to T_QC in "pp.h"). The symbols "QC" and "CPM" are both predefined to "1".

4. **Z80 code to run under CP/M, compiled under MS-DOS** (written using the QCX cross compiler, configuration option TARGET set to T_QCX in "pp.h"). The symbols "QC" and "CPM" are both predefined to "1".

5. **Transputer code** (written using the TCX cross compiler, configuration option TARGET set to T_TCX in "pp.h"). The symbol "TC" is predefined to "1".

6. **Generic SYS5 UNIX systems** (configuration option TARGET set to T_UNIX in "pp.h"). The symbol "unix" is predefined with a value of "1".

7. **VAX VMS systems** (configuration option TARGET set to T_VMS in "pp.h"). The symbol "VMS" is predefined to be "1".

8. **XENIX systems** (configuration option TARGET set to T_XENIX in "pp.h").

PP Default Include File Pathname

The default include file pathname used by PP is determined by the definition of DFLT_PATH in "pp.h". This in turn is dependent on the setting of the HOST configuration definition (also in "pp.h"). The possible settings are:

1. **BSD 4.X UNIX systems** (configuration option HOST set to H_BSD). The default include file pathname (DFLT_PATH), is set to "/usr/include".

2. **CP/M systems** (configuration option HOST set to H_CPM). The default include file pathname (DFLT_PATH), is set "A0" and the use of the pathname file PATHFILE (defined in "pp.h"), is supported. The PATHFILE definition is set to "path.ppi" in the release versions of PP (see the description of the "-i" command line option in the **Option Descriptions** section).

3. **Mac's running MPW** (configuration option HOST set to H_MPW in "pp.h"). The default include file pathname (DLFT_PATH), is set to "TIncludes".

4. **MS-DOS systems** (configuration option HOST set to H_MSDOS). The default include file pathname (DFLT_PATH), is set to "/include".

5. **Generic SYS5 UNIX systems** (configuration option HOST set to H_UNIX). The default include file pathname (DFLT_PATH), is set to "/usr/include".

6. **VAX VMS systems** (configuration option HOST set to H_VMS). The default include file pathname (DLFT_PATH), is set to "crel$include:".

7. **XENIX systems** (configuration option HOST set to H_XENIX in "pp.h"). The default include file pathname (DLFT_PATH), is set to "/usr/include".

Non-Standard Features of PP

Conforming to the standard is all well and good, but extensions are really where the fun is! PP includes an extended form of macro parameter processing:

When defining a macro or #define, an optional form of formal parameter definition is allowed. If the formal parameter is enclosed in square brackets, then optional director fields are placed after the formal name to control the manner in which the actual parameter is bound to the formal at invocation. PP currently supports either "RQ" or "RN" as director fields. RQ causes any "" characters (surrounding a string) to be removed at the time the parameter is bound to the formal parameter name. For example:

```
#pragma macro  asm([line,RQ])
;
#pragma   asm
line
#pragma   endasm
#pragma   endmacro
```

In fact, the above IS the internal definition of the asm() function. This function is provided on many "C" compilers as a more dignified way of doing inline assembly language processing than "#asm"/"#endasm". Thus, the "asm()" function is converted by PP using the above macro INTO "#asm"/"#endasm" pairs and included code! For example:

```
asm("ADD");

This is written to the output file by PP as:

;
#asm
ADD
#endasm
```

In contrast to "RQ", the "RN" director causes any newlines contained in the corresponding macro parameter to be replaced with blanks during invocation. This is useful when macro arguments are to be used with other PP facilities which are line oriented (such as "#if").

Finally, two preprocessing "pseudo" functions have been added to PP to facilitate compile time string processing optimization. These functions are used in a similar fashion to "defined", except they require parenthesis around the argument:

```
_isstring()
```

This function evaluates to TRUE if its argument is a "C" string constant. If not it evaluates to FALSE.

`_strsize()`

This function requires that its parameter be a "C" string constant (or several adjacent string constants). It evaluates to the amount of storage in bytes which will be required to hold the composite string (including the terminating '\0').

The above two PP functions, together with the "#pragma value" directive allows a modest amount of compile time optimization in choosing which runtime function (if any), is needed to evaluate a particular string operation. For example:

```
#pragma   macro     strlen([x,RN])
#pragma   if   _isstring(x)
((size_t)
#pragma   value     (_strsize(x) - 1)
)
#pragma   else
strlen(x)
#pragma   endif
#pragma   endmacro
```

This removes the call to the "strlen" routine if the result is known at compile time. Similar techniques can be used to conditionally convert "strcpy" calls into "memcpy", etc.

Appendix A: Error Messages

Types of Error Messages

There are three classes of error messages which PP can generate:

1. **Warnings.** The general form of these informative messages is:

```
<filename> @ line_number: WARNING: message_text
```

Where the "<filename>" field indicates the current source code file being read, the "line_number" field gives the line where the problem was detected, and the "message_text" field indicates the actual problem encountered.

2. **Non-fatal errors.** These are used for reporting actual error conditions which will affect the error count PP keeps, and the error status of the return value given when PP exits. The exit error status set by encountering one or more non-fatal errors may be overridden by using the "-e" command line option flag to indicate you do not wish to abort on errors. The format for non-fatal errors is:

```
<filename> @ line_number: message_text
```

Where "<filename>", "line_number" and "message_text" have the same meaning as in the "warning" message description above.

3. **Fatal errors.** If the problem detected by PP is so severe that it can't continue operating, it will give a "fatal" error message:

```
FATAL: message_text
```

After printing one of these messages, PP will immediately exit with its error return code set. This happens regardless of whether the command line contained a "-e" option flag or not.

In addition to the above warning and error messages, the user may initiate two forms of messages using pragma directives:

4. **Normal non-fatal error messages with custom "message_text" fields.** These are generated using the "#pragma error message_text" directive and are treated in all aspects as if the error was detected by PP itself.

5. **Informative messages using the "#pragma message message_text" directive.** This message doesn't signal an error and is not treated as such by PP. The output format is:

```
<filename> @ line_number: MESSAGE: message_text
```

Error Message Descriptions

The following descriptions list the various warning and error messages which PP can generate (in alphabetic order):

```
<filename> @ line_number: Already within "#pragma asm"
```

This error occurs when another "#pragma asm" is encountered while still within a previous one. These critters can't be nested.

```
<filename> @ line_number: Bad directive
```

On the specified input filename, at the specified line #, a bad preprocessor directive was encountered (a "#" on a line by itself?).

```
<filename> @ line_number: Bad include argument
```

The filename parameter to a "#include" was not in correct form.

```
<filename> @ line_number: Bad parameter to "#error"
```

The string display parameter to a "#error" was not in correct form.

```
<filename> @ line_number: "#elif" outside of "#if"
```

This error indicates a "#elif" directive was found outside the scope of a "#if"/"#endif" conditional compilation statement.

```
<filename> @ line_number: "#else" already encountered
```

This error indicates a "#else" directive was encountered in a conditional compilation statement which already contained a "#else" within the same scope.

`<filename> @ line_number: "#else" outside of "#if"`

This error indicates a "#else" directive was found outside the scope of a "#if"/"#endif" conditional compilation statement.

`<filename> @ line_number: "#endif" outside of "#if"`

This error indicates that a "#endif" was encountered that had no matching "#if".

`<filename> @ line_number: EOF in comment`

This error indicates an end of file was encountered while processing a comment.

`<filename> @ line_number: Expression: Bad operand: token`

The listed preprocessor token is not a valid operand at this point in the evaluation of a preprocessor constant expression.

`<filename> @ line_number: Expression: Division by zero`

A division or remainder operator in a preprocessor constant expression had zero as the right operand.

`<filename> @ line_number: Expression: EOL in '' constant`

This error is generated during the evaluation of a preprocessor constant expression when the end-of-line character is detected within a character constant.

`<filename> @ line_number: Expression: ':' expected`

This error is generated during the evaluation of a preprocessor constant expression when a ternary operator is being processed ("?:"), and the colon is not where it should be.

```
<filename> @ line_number: Expression: Expected operand: token
```

The listed preprocessor token was expected in the constant expression expression at some point and was not found.

```
<filename> @ line_number: Expression: Illegal character escape
```

This indicates that a illegal character followed the backslash escape character in a character constant expression. This is detected during the evaluation of preprocessor constant expressions.

```
<filename> @ line_number: Expression: Illegal hex digit
```

This indicates that an illegal character was found in a hex character constant. This is detected during the evaluation of preprocessor constant expressions.

```
<filename> @ line_number: Expression: Illegal octal digit
```

This indicates that an illegal character was found in a octal character constant. This is detected during the evaluation of preprocessor constant expressions.

```
<filename> @ line_number: Expression: Invalid operator: operator
```

This error is generated during the evaluation of a preprocessor constant expression when a unexpected token/symbol/operator is encountered.

```
<filename> @ line_number: Expression: Mismatched "()"
```

This error is generated during the evaluation of a preprocessor constant expression when a matching ")" is not present for a previous "(".

```
<filename> @ line_number: Expression: Mismatched apostrophes
```

This error is detected during preprocessor constant expression evaluation when a character constant is encountered (an opening apostrophe), without a closing, matching, apostrophe.

`<filename> @ line_number: Expression: Missing '('`

This error is generated when a pseudo function such as "_isstring" is used within a preprocessor constant expression and the leading "(" is missing.

`<filename> @ line_number: Expression: Missing ')'`

This error is generated when a pseudo function such as "defined()" is used within a preprocessor constant expression and the trailing ")" is missing.

`<filename> @ line_number: Expression: Not an identifier: macro_symbol_name`

This error is generated when the macro symbol name used in a "defined()" pseudo function is not a legal symbol name.

`<filename> @ line_number: Expression: sizeof() not allowed`

This error message is generated during the evaluation of a preprocessor constant expression when the "sizeof()" pseudo function is referred to. This function is not implemented in PP. The ANSI "C" standard specifically prohibits the use "sizeof" in preprocessor constant expressions.

`<filename> @ line_number: Expression: Token too long`

A token was encountered during preprocessor constant expression evaluation which was too long for PP. The limit definition for this is TOKENSIZE in "pp.h".

`<filename> @ line_number: _strsize: Missing string`

This happens when the PP "pseudo" function "_strsize" is called with something other than a "C" string constant as an argument. You MUST use the "_isstring" function and a "#if" to ensure that "strsize" is only called with a constant string as an argument. This specific error message is generated when the first parameter isn't a string.

`<filename> @ line_number: _strsize: Not a string`

This happens when the PP "pseudo" function "_strsize" is called with something other than a "C" string constant as an argument. You MUST use the "_isstring" function and a "#if" to ensure that "strsize" is only called with a constant string as an argument. This specific error message is generated when the first parameter was a legal string, but some subsequent token, which was encountered before the terminating ")", wasn't a legal string constant.

`FATAL: Bad option: option_string`

The option_string you used was incorrect in some fashion.

`FATAL: Input and output filenames are the same: filename`

You can't use the same "filename" for both the input and output.

`FATAL: Out of memory`

This error message is generated whenever any of the PP functions which use "malloc" to get memory are unable to obtain the amount needed.

`FATAL: Pushback buffer overflow`

This error message happens when the "pushback" buffer (used during macro expansion, etc.), overflows. The buffer size is defined by PUSHBACKSIZE defined in "pp.h".

`FATAL: Unable to create output file: filename`

Unable to open/create the output "filename".

`FATAL: Unable to close output file: output_filename`

This error is detected when PP is finished and attempts to close the output file.

```
FATAL: Unable to open input file: filename
```

The open attempt for the input "filename" failed.

```
FATAL: Unexpected EOF
```

This error occurs when EOF is encountered in some place where it really doesn't belong (say within the parameter list for a macro, or somewhere like that).

```
FATAL: Too many pathnames
```

Generated if too many command line "-i" switches were given. Remember that the configuration include file path and any paths specified in the "pathname" file also count against this limit.

```
<filename> @ line_number: "#if" stack overflow
```

The nesting level of conditional compilation directives was exceeded. The limit is defined by IFSTACKSIZE in "pp.h".

```
<filename> @ line_number: Illegal directive
```

On the specified input filename and line # an illegal preprocessor directive was found.

```
<filename> @ line_number: Illegal disk drive specifier
```

Only used by the CP/M version of PP. Indicates the drive name used in a include file pathname specification was not in the range of "a" to "p".

```
<filename> @ line_number: Illegal file name
```

The filename for a "#include" was in bad form.

```
<filename> @ line_number: Illegal or out of place token: token_string
```

This error message is generated when a unexpected token type is encountered during the parsing of the parameters for a macro.

`<filename> @ line_number: Illegal redefinition of symbol name: macro_symbol_name`

You get this if you redefine an existing macro symbol to a new value. This error can only occur if you are using the "non-stacking" model of macro definitions (ANSI), where each new definition is either identical to the previous definition (and ignored), or different (causing this error message).

`<filename> @ line_number: Illegal symbol name: macro_symbol_name`

This error message is generated by malformed macro symbol names.

`<filename> @ line_number: Illegal user number`

Only used by the CP/M version of PP. Indicates the user # used with a include file pathname specification was not in the range of 0 to 31.

`<filename> @ line_number: Include filename too long`

Self explanatory. The limit is defined by FILENAMESIZE in "pp.h".

`<filename> @ line_number: Include file stack overflow`

If too many nested include files are used you get this error. The limit value is defined by FILESTACKSIZE in "pp.h".

`<filename> @ line_number: Invalid formal parameter`

This error message is generated when you specify a formal flag trailing a [] formal parameter block which is not begun by a alpha. See the **Non-Standard Features of PP** section for more information about this extension.

`<filename> @ line_number: Invalid formal parameter flag: flag_name`

This error message is generated when you specify a formal flag trailing a [] formal parameter block which is not implemented. At the moment, "RQ" is the only legal flag string (see the **Non-Standard Features of PP** section for more information about this extension).

`<filename> @ line_number: "#line" argument error`

An error was encountered when an incoming "#line" preprocessor directive was being parsed.

`<filename> @ line_number: Macro body overflow`

PP was unable to fit the macro body text into the expansion buffer. This is generally caused by recursive or deeply nested macro calls.

`<filename> @ line_number: Macro too long`

PP was unable to fit the macro body text into the expansion buffer. This is generally caused by recursive or deeply nested macro calls.

`<filename> @ line_number: Not within "#pragma asm"`

This error occurs when a "#pragma endasm" is encountered without a matching "#pragma asm" having been encountered first.

`<filename> @ line_number: Parameter buffer overflow`

PP was unable to fit the macro parameter token into the parameter buffer. This is generally because the parameter was itself the result of a macro which expanded in size greatly.

`<filename> @ line_number: "#pragma endmacro" illegal outside macro`

This error occurs when a "#pragma endmacro" is encountered without a matching "#pragma macro" directive.

`<filename> @ line_number: Read buffer overflow`

The input line read by PP was too large to fit into the corresponding buffer for that token type.

`<filename> @ line_number: Token too long`

A token was encountered which was too long for PP. The limit definition for this is TOKENSIZE in "pp.h".

`<filename> @ line_number: Unable to close include path file: path_file`

This error message can only occur on CP/M versions of PP. It happens when PP is closing "path_file" after reading the desired include file search pathnames.

`<filename> @ line_number: Unable to close input/include file: filename`

This error is generated whenever a input source or include file gets an error when it is closed.

`<filename> @ line_number: Unable to open include file: filename`

The specified "filename" can't be found on any of the provided include file search paths.

`<filename> @ line_number: Unterminated conditional`

On the specified input file, at the specified line #, the end of the input file was reached without a closing "#endif" for a preprocessor conditional compilation directive.

```
<filename> @ line_number: WARNING: Bad format on include path file:
path_file
```

This error message is only used on CP/M versions of PP, when the include file paths specified on "path_file" aren't decipherable ("path_file" defaults to "path.ppi" in the release version). When this happens the default configuration pathname is used in place of whatever "path.ppi" was trying to specify.

```
<filename> @ line_number: WARNING: "/*" found in comment
```

This is just a warning message indicating that a "/*" was detected while hunting for a "*/" comment terminator. This message is only possible if recursive comments have been disabled (see RECURSIVE_COMMENTS in appendix B).

```
<filename> @ line_number: WARNING: Incorrect argument count for:
macro_symbol_name
```

Generated when you use a parameterized macro and supply the wrong number of arguments.

```
<filename> @ line_number: WARNING: Redefining symbol: macro_symbol_name
```

You get this if you redefine an existing macro symbol to a new value. This error can only occur if you are using the "stacking" model of macro definitions (non-ANSI), where each new definition hides the previous definition until a subsequent "#undef" agains brings it to view.

```
<filename> @ line_number: WARNING: Symbol already defined: macro_symbol_name
```

If a command line symbol defined via a "-d" option flag is given which conflicts with an existing macro symbol name, this error message is given and the new definition is ignored.

```
<filename> @ line_number: WARNING: Symbol not defined: macro_symbol_name
```

If you attempt to un-define a non-existant predefined macro symbol you get this warning.

Appendix B: PP Internals

Source Code Organization and Compiling

The "C" source code for PP is contained in two include files ("pp.h" and "ppext.h"), and eight source code files ("pp1.c" through "pp8.c"). To compile the program you compile all eight source code files and link the result together to form the PP executable (see the notes in the next section if you are compiling the CP/M version). If you intend to do any significant work on PP it is recommended that you set up a "makefile" file for the preprocessor with the dependencies set such that a change in the include file causes all the source code files to be recompiled, but that changes to individual source code files do not force the recompiling of the others. A sample "makefile" file is provided for use with the supplied MAKE utility and Microsoft 'C' version 6.00a or Borland C V2.0. The "makefile" file will also be useful with most UNIX versions of MAKE assuming the compile/link commands are changed to whatever the local environment requires. If you do not have a MAKE facility you should at least set up a batch/command file to compile PP (too many mistakes otherwise)!

The following is a brief description of the contents of the source code files:

`"pp.h"`

The main PP "C" include file. Contains the macro definitions for the configuration options and the external structure and data definitions. Also, contains the macro definitions which are used to make the various internal PP states and data information "symbolic". Note that this include file is used by all eight source code files, but it behaves differently when it is included by "pp1.c" than when the other files include it. This is caused by a "MAIN" symbol which is defined in "pp1.c" and not in the others. This allows data definitions for "pp1.c" (where global data is defined), to be defined in "pp.h".

`"ppext.h"`

This include file contains external function definitions for the functions which make up PP. This file is included in all eight source files.

`"pp1.c"`

This source code file contains most of the global data definitions (see "pp.h"), and also the command line interface code for PP.

`"pp2.c"`

This source code file contains the higher level code for handling preprocessor macros and the preprocessor symbol table.

`"pp3.c"`

This source code file contains the general I/O functions used by PP, including both external file and internal buffer I/O. Note that the read character function "nextc" which is used extensively is not in this file, moreover it doesn't exist! In actual fact, it is a macro definition for a function pointer (defined in "pp.h"), which points to the actual function desired (for instance "gchbuf" in "pp3.c"). In a similar vein, the "achar" function defined in this file is replaced by a macro definition in "pp.h" if the PP preprocessor symbol in "pp.h" is TRUE (see the next section and the source code for more information).

`"pp4.c"`

This source code file contains the general preprocessor token processing and classification routines. Note the "istype" function is replaced by a macro definition in "pp.h" if the PP preprocessor symbol in "pp.h" is TRUE (see the next section and the source code for more information).

`"pp5.c"`

This source code file contains the conditional compilation code for PP.

`"pp6.c"`

This source code file contains the "pragma" processing code for PP.

`"pp7.c"`

This source code file contains the error message processing routines for PP.

`"pp8.c"`

This source code file contains the constant expression evaluation code for PP.

`"ppbanner.c"`

This source code file is only needed when compiling PP with the HOST definition set to H_CPM, see the next section for more details.

Configuring the PP Source Code

There are two areas (at least), of the "pp.h" file which need to be changed if you plan on moving PP to another operating environment. You need to set the "HOST" macro definition to one of: H_BSD, H_CPM, H_MPW, H_MSDOS, H_UNIX, H_VMS or H_XENIX. You also need to specify the "TARGET" environment for the "C" code which PP is compiling: T_BSD, T_MPW, T_QC, T_QCX, T_TCX, T_UNIX, T_VMS or T_XENIX. As released, the CP/M version of PP is set to HOST == H_CPM and TARGET == T_QC. The MS-DOS version of PP for use with either the QCX or TCX cross compilers is set to HOST == H_MSDOS and TARGET == T_QCX (or T_TCX as appropriate). Note that if you select HOST == H_CPM (as in the CP/M release), you must provide an additional file ("ppbanner.c"), to be linked in with the eight source files. "ppbanner.c" contains a single data structure definition of the form:

```
        char *ver_id="PP Vxxx";
```

This line defines the startup banner for PP to print to the user. In development work under CP/M this file is automatically emitted by the "make" utility and includes the current data and time. If you select any other host environment other than H_CPM this file is not required and you need to edit the VERSION macro definition in pp.h to change the banner value.

The most generic of the configuration options are the HOST == H_UNIX and TARGET == T_UNIX selections. These assume "C" standard library routines and are a good first step when moving PP. Note that the "TARGET" configuration options are primarily used to select what "predefined" preprocessor definitions are generated. If you select T_UNIX you get the predefined symbol "unix". If you select T_BSD you get "unix" and "BSD". If you select T_VMS you get "VMS". If you select T_QC or T_QCX you get "QC" and "CPM". If you select T_TCX you get "TC".

Another function of the "TARGET" selection is to include facilities in PP which are required for particular "C" compilers. For instance, the QC/QCX/TCX compilers allow inline assembly language which is done via PRAGMA's and is only enabled if TARGET is set to T_QC, T_QCX or T_TCX. See the source code (particularly "pp.h") for additional information about this stuff. Note that not all combinations of the supplied configuration options have been tested and that minor problems may arise with some.

Another configuration option which may need to be changed is the PP_SYSIO macro definition in "pp.h". If this macro is defined PP is compiled to use the lowest level character I/O available from "C" ("open", "read", etc.). This level of I/O tends to be less portable between different compilers and machines than the higher level stream I/O used if PP_SYSIO is not defined. Note that which version of the PP_SYSIO macro needs to be modified in "pp.h" depends on the setting of the HOST configuration options. As a concrete example of this the Lattice MS-DOS compiler version 2.15E would not run correctly with PP_SYSIO defined. Changes have since been made to PP which might make it more robust to whatever the problem was (or Lattice may have improved things with newer versions). In any event, when moving PP you might want to comment out the appropriate PP_SYSIO definition initially.

Like all the above, the DFLT_PATH configuration option is a macro defined in "pp.h". It allows you to specify the default search path to use when PP encounters a "#include <XXX>" statement. The default pathname is appended in front of the "XXX" filename to determine where to read the include file from. Like PP_SYSIO, the particular DFLT_PATH which you need to change in "pp.h" depends on the HOST configuration parameter. Note that if the HOST == H_CPM configuration option is selected the situation is much different. Because CP/M 2.2 (assumed with HOST == H_CPM), doesn't support a tree structured directory drive/user-number pairs are used instead. Another feature added to the CP/M version is the existance of a default "path" file: What happens is that when PP starts up it looks on the drive/user-number pair specified in DFLT_PATH to attempt to read a file whose name is defined by another macro definition in "pp.h" which is named PATHFILE. The contents of this file (named "path.ppi" in the CP/M release), are read and interpreted as if they were a series of "-i" include pathname specifiers on the PP command line. See the "-i" command line option information in section "PP Syntax & Semantics" for more information about include file search paths.

The ENV_PATH configuration option is a macro defined in "pp.h" which is used to allow DFLT_PATH to be supplanted by a user specified environment variable. The environment variable to be used is specified by the value of ENV_PATH. See the "-i" command line option information in section "PP Syntax & Semantics" for more information about include file search paths.

If you are moving PP to another machine you should make sure that the DEBUG definition in "pp.h" is set to TRUE. This causes code to be included which allows the "-z" command line flag to output debugging information to the console when PP is running. The DEBUG flag is enabled in the release versions of PP and it should probably stay enabled unless severe space restrictions force otherwise. Note that the debug messages are somewhat cryptic, but they really do help narrow problems down! See the source code for the possible messages and their meaning (if you are debugging changes you've made you already know more than I can tell you about this sort of stuff)!

If the "C" compiler you are using to compile PP doesn't handle parameterized macro's, but does handle most other preprocessor facilities, you can set the PP macro definition (in "pp.h"), to be FALSE. This causes all instances of parameterized macro's to be replaced with actual function calls. Although this is a somewhat esoteric configuration option, it was required with the original version of QC which was used to develop PP.

Another "major" configuration option: The VERBOSE macro definition in "pp.h" selects whether PP is verbose by default or not. If verbose is defined to be FALSE, then PP is quiet by default and the "-v" switch has to be thrown to make it verbose. If VERBOSE is TRUE, then PP is verbose by default and throwing "-v" makes it quiet.