

PreSoC *A Rational Preprocessor*

Author: Dave Van Ess
Associated Project: Yes
Associated Part Family: All
PSoC Designer Version: All
Associated Application Notes: None

Abstract

Sometimes it is necessary to write assembly code. A preprocessor can add logical control structures and more naturally formatted instructions, allowing assembly code to appear more “C” like. This Application Note includes the source code and executable file for a PSoC™ preprocessor.

Introduction

A pivotal work in the development of software, as an engineering discipline, is Kernighan and Plauger’s book, “Software Tools,” first published in 1976. At the time of publication, C was a language available only at Bell Laboratories and a few universities. They wisely chose to use FORTRAN for their examples, as it was widely used and accepted in industry. FORTRAN’s main problem was that it lacked the control structures that they were intending to teach. To alleviate this problem, the authors created a preprocessor for FORTRAN that added many of the C control structures to the FORTRAN language. They named this preprocessor “**Rational Fortran**” or **RATFOR** for short.

This work had several effects. The first being that this led to the acceptance of C and the transformation of established languages to include logical control structures. Ten years later, C became the de facto standard for writing control software. Companies not converting over to a C development environment limited their ability to hire competent software engineers. The second effect being that other languages quickly added logical control structures or risked becoming obsolete.

The third effect was to encourage others to write preprocessors for languages other than FORTRAN.

It was only natural that some preprocessors would be written for assembly language. A programmer could now customize their assembler to fit their requirements.

Assemblers have not changed much and assembly code has become the output of C compilers. Assembler development is not considered a highly coveted assignment. However, the need for structured assembly code still exists. PreSoC is a preprocessor written for the PSoC assembly language.

Assembly Control Structures

The PSoC assembly language has five instructions used for conditional control. They are:

- o `jc` Jump if Carry flag is set
- o `jnc` Jump if Carry flat is not set
- o `jz` Jump if Zero flag is set
- o `jnz` Jump if Zero flag is not set
- o `jacc` Jump on Accumulator

The first four are the most widely used. A code example is shown as follows.

```
Mov [LoopCount], 10
StartOfLoop:
    dec [LoopCount]
    jnz StartOfLoop
```

Code 1.

This example decrements the variable until its value becomes zero.

For the PSoC preprocessor, 22 logical control instructions have been added. They are:

```

o  if_z
o  if_nz
o  if_c
o  if_nc
o  else_if_z
o  else_if_nz
o  else_if_c
o  else_if_nc
o  else_
o  end_if
o  while_z
o  while_nz
o  while_c
o  while_nc
o  end_while
o  repeat
o  until_z
o  until_nz
o  until_c
o  until_nc
o  loop
o  end_loop

```

Using these instructions, Code 1 is rewritten and shown as Code 2.

```

mov [LoopCount], 10
repeat
  dec [LoopCount]
until_z

```

Code 2.

Of course these new structures are made up of the actual assembly instructions. The preprocessor adds the necessary labels and substitutes the required instructions. This detail is just hidden from the user.

if_, else_if, else_, end_if Construct

This construct takes the following form:

```

if_x ; x is z,nz,c,or nc
;Code
else_if_x
;Code
else_
;code
end_if

```

Code 3.

There are several rules for using this construct.

- o Only one `else_if_x` is allowed.
- o It must follow `if_`.
- o Only one `else_` is allowed.
- o It must follow either `if_x` or `else_if_x`.
- o `end_if` ends the construct.
- o It must follow `if_x`, `else_if_x`, or `else_`.

When decoding this structure, the preprocessor generates unique labels.

while_, end_while Construct

This construct takes the following form:

```

;1st decision flags set here
while_x ; x is z,nz,c,or nc
;Code
;flags set here
end_whilef

```

Code 4.

There are several rules for using this construct.

- o Only one `end_while` is allowed.
- o It must follow `while_x`.

The fact that the flags are set in two different points limits the usefulness of this construct. This is addressed in the next two constructs.

repeat, until_ Construct

This construct takes the following form:

```

repeat
;Code
until_x ; x is z,nz,c,or nc

```

Code 5.

There are several rules for using this construct.

- o Only one `until` is allowed.
- o It must follow `repeat`.

loop, end_loop Construct

This construct takes the following form:

```

loop
;Code
jx OutOfLoop ; exitif_x
end_loop
OutOfLoop:

```

Code 6.

There are several rules for using this construct.

- o Only one `end_loop` is allowed.
- o It must follow `loop`.

This loop is continuous. No `exit_if_x` command exists. The user must fabricate it with a convention branch instruction.

Why does Assembly have to look so Assembly?

Assembly language instructions generally take this form:

```
LABEL: OPCODE DESTINATION, SOURCE ;Comment
```

Code 7.

Assembly started as a shortcut to writing machine code. The format is hardware centric and was done in such a manner as to make the parsing easier. The original assemblers were done on computers with very, very little memory. This instruction format has persisted because this is what assembly is supposed to look like. Humans resist change. (A clock's hands move in the same direction as a sundial in the northern hemisphere. Knobs turn clockwise for increased amplitude because civilization started 10,000 years ago in the Northern Hemisphere.)

The PreSoC preprocessor used the following format for instructions.

```
LABEL:
DESTINATION OPERATION SOURCE; Comment
```

Code 8.

This format is parsed and converted to the standard form used by the assembler. The different instructions, and conversions, are shown in Table 1.

Any instruction not having this format is assumed to be an actual assembly instruction, macro, or directive and is not changed.

Using PreSoC

PreSoC is a three-pass preprocessor with each pass doing the following:

First Pass

- o Places a carriage return after the first label on a line not followed by an `equ` or `blk`.
- o Removes any left justified white space.
- o Adds a comment to the end of the line with the line number of the code.

The last step is for debugging purposes. The output of PreSoC is an assembler program. Any mistakes in the program can be referenced to the original preprocessor file via these line numbers.

Table 1. Instructions and Conversions

Preprocessor Format	Assembly Format
<code>dest += source</code>	<code>add dest, source</code>
<code>dest +=c= source</code>	<code>adc dest, source</code>
<code>dest -= source</code>	<code>sub dest, source</code>
<code>dest -b= source</code>	<code>sbb dest, source</code>
<code>dest ++</code>	<code>inc dest</code>
<code>dest --</code>	<code>dec dest</code>
<code>dest <<</code>	<code>asl dest</code>
<code>dest >></code>	<code>asr dest</code>
<code>dest <<<</code>	<code>rlc dest</code>
<code>dest >>></code>	<code>rrc dest</code>
<code>dest &= source</code>	<code>and dest, source</code>
<code>dest = source</code>	<code>or dest, source</code>
<code>dest ^= source</code>	<code>xor dest, source</code>
<code>dest <=? source</code>	<code>cmp dest, source</code>
<code>dest &? source</code>	<code>tst dest, source</code>
<code>dest = source</code>	<code>mov dest, source</code>
<code>dest = [source ++]</code>	<code>mvi dest, source</code>
<code>[dest ++] = source</code>	<code>mvi dest, source</code>
<code>dest <-> source</code>	<code>swap dest, source</code>

Second Pass

The second pass takes the output of the first pass and parses the logical control structures into assembly instructions. It checks to see that the rules for each construct have been met. When violations are found, the preprocessor flags the user of the problem and stops. If no rule violations are found, the second pass passes its output to the third pass.

Third Pass

The third pass looks for any instances of the new instructions and converts them to the form acceptable to the assembler. The output of this pass is the assembly file.

Example

For this example, a preprocessor file is created in a PSoC project as a text file. A text file can be edited with the editor supplied with PSoC Designer. `testfile.tx` is shown in Figure 1.

```

testfile.txt
;Test Program
;sum the absolute value
;of 24 signed integers.
;
include "m8c.inc"
area bss(RAM)
    iTotal:   BLK 2   ;
    bCount:   BLK 1   ;
area text(ROM,REL)

export TestRoutine
export _TestRoutine
LowByte: equ 1
HighByte: equ 0

TestRoutine:
_TestRoutine:
    [iTotal + HighByte] = 0
    [iTotal + LowByte] = 0
    [bCount] = 23
    repeat
        call iGetdata ;data returned in X,A
        A <-> X
        A <=? 80h      ;c set if msbit is zero
        A <-> X
        if_c ;positive
            [iTotal + LowByte] += A
            A = X
            [iTotal + HighByte] +c= A
            [bCount] --
        else_ ;negative value
            [iTotal + LowByte] --= A
            A = X
            [iTotal + HighByte] -b= A
            [bCount] --
        end_if
    until_c;done
Alldone: ret

```

Figure 1. testfile.txt Preprocessor File

Now use Windows Explorer to open the project file and place there a copy of *PreSoc.exe*. The directory is shown in Figure 2.

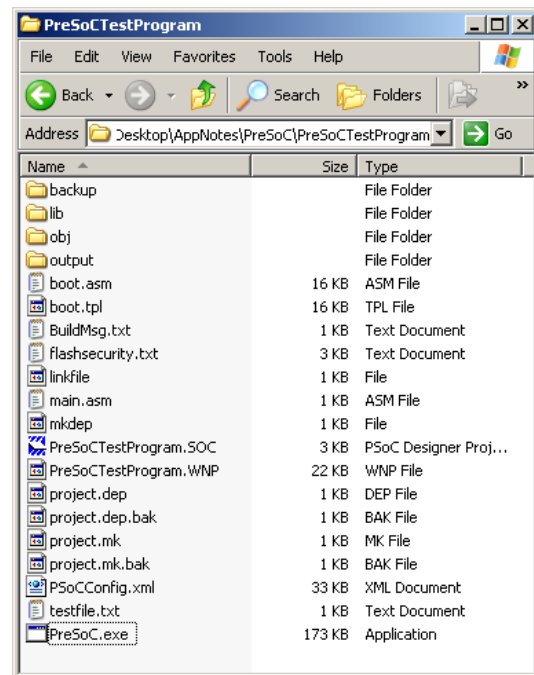


Figure 2. Project Directory with "testfile.txt" and "PreSoC.exe"

While in this file, double-click on *PreSoc.exe* to start up the program. Figure 3 shows the resulting console window.

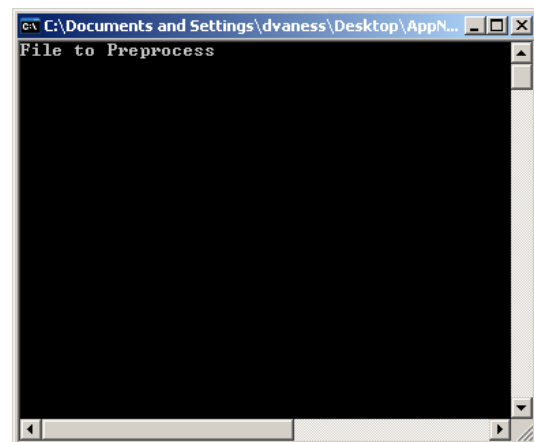


Figure 3. PreSoC Console Window

Enter the file name to be preprocessed *testfile.txt*<cr>. The three passes are then run. The console window is again shown in Figure 4.

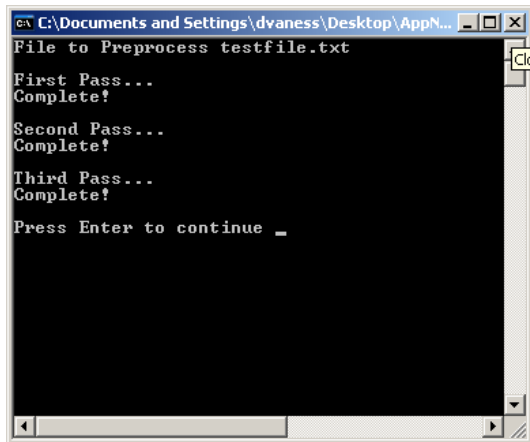


Figure 4. Successful Preprocessing Complete

“Enter” closes the window. From PSoC Designer or Windows Explorer, the file can be examined. Figure 5 shows the window used to select a file for PSoC Designer.

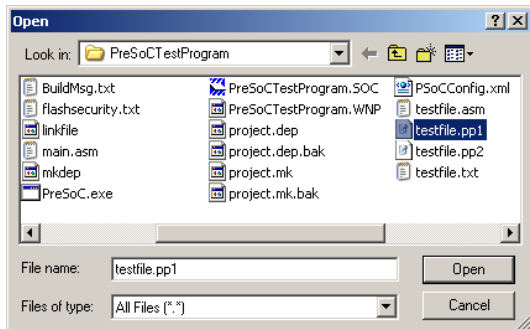


Figure 5. Open testfile.pp1

The output of the first pass is shown in Figure 6.

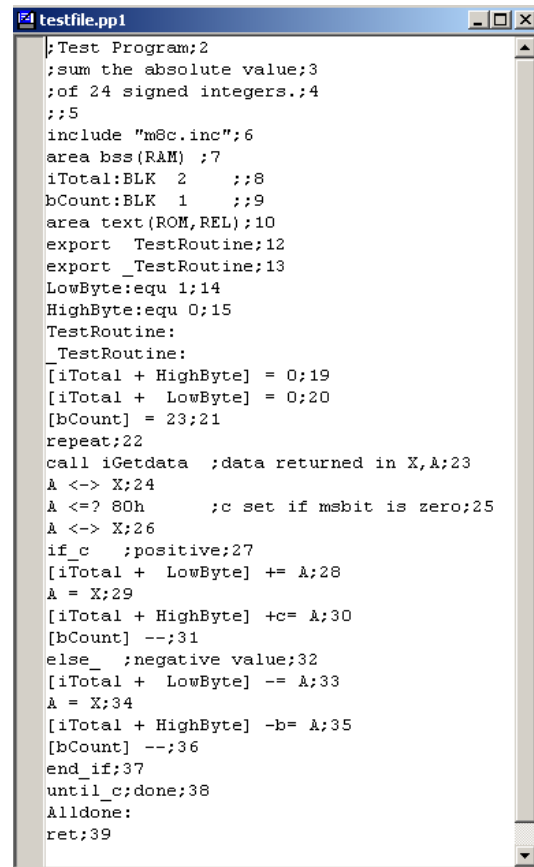


Figure 6. First Pass Results testfile.pp1

Note that the white space has been removed and line numbers have been appended, as comments, to the end of each line.

The output of the second pass is shown in Figure 7.

```

testfile.pp2
;Test Program;2
;sum the absolute value;3
;of 24 signed integers.;4
;;5
include "m8c.inc";6
area bss(RAM) ;7
iTotal:BLK 2 ;;8
bCount:BLK 1 ;;9
area text(ROM,REL);10
export TestRoutine;12
export _TestRoutine;13
LowByte:equ 1;14
HighByte:equ 0;15
TestRoutine:
_TestRoutine:
[iTotal + HighByte] = 0;19
[iTotal + LowByte] = 0;20
[bCount] = 23;21
repeat_1::22
call iGetdata ;data returned in X,A;23
A <-> X;24
A <=? 80h ;c set if msbit is zero;25
A <-> X;26
if_2:jnc elseif_2 ;positive;27
[iTotal + LowByte] += A;28
A = X;29
[iTotal + HighByte] += A;30
[bCount] --;31
elseif_2:
jmp endif_2
else_2: ;negative value;32
[iTotal + LowByte] -= A;33
A = X;34
[iTotal + HighByte] -= A;35
[bCount] --;36
endif_2::37
until_1:jnc repeat_1;done;38
Alldone:
ret;39

```

Figure 7. Second Pass Results testfile.pp2

The logical control constructs have all been successfully converted to actual assembly instructions.

Figure 8 shows the output of the third and final pass.

```

testfile.asm
;sum the absolute value;3
;of 24 signed integers.;4
;;5
include "m8c.inc";6
area bss(RAM) ;7
iTotal:BLK 2 ;;8
bCount:BLK 1 ;;9
area text(ROM,REL);10
export TestRoutine;12
export _TestRoutine;13
LowByte:equ 1;14
HighByte:equ 0;15
TestRoutine:
_TestRoutine:
mov [iTotal + HighByte],0;19
mov [iTotal + LowByte],0;20
mov [bCount],23;21
repeat_1::22
call iGetdata ;data returned in X,A;23
swap A,X;24
cmp A,80h ;c set if msbit is zero;25
swap A,X;26
if_2:jnc elseif_2 ;positive;27
add [iTotal + LowByte],A;28
mov A,X;29
adc [iTotal + HighByte],A;30
dec [bCount] ;31
elseif_2:
jmp endif_2
else_2: ;negative value;32
sub [iTotal + LowByte],A;33
mov A,X;34
sbb [iTotal + HighByte],A;35
dec [bCount] ;36
endif_2::37
until_1:jnc repeat_1;done;38
Alldone:
ret;39

```

Figure 8. Third Pass Results of testfile.asm

The instructions have all been converted back to actual assembly instructions. This file can be successfully assembled.

For the project to recognize this file it has to be added to the project. It is done so by going to: **Project >> Add To Project >> Files**

The contents of this directory are in the dialog box shown in Figure 9.

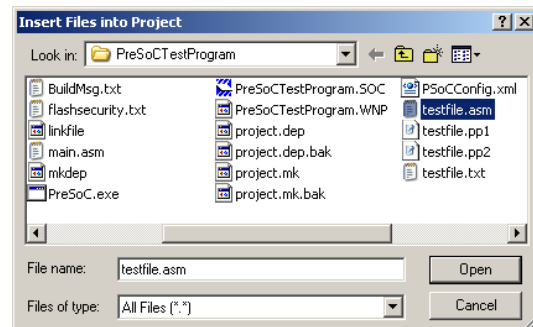


Figure 9. Adding testfile.asm to Project.

This file is now available to the project. Changes can be made to it or to the original file. However, changes made to the original preprocessor file require running the preprocessor.

How May I Get a Copy of PreSoC

It is included in the project file accompanied with this Application Note. It is written in C. Both the source code and executable files are included. The source is included because PreSoC is open source software. Copies are also available at www.PsoCDeveloper.com. Other users are encouraged to make additions and modifications and submit them to the previously mentioned site. Possible improvements include:

- A better user interface.
- The adding of a `define` feature.
- More than one `else_if` for each `if_`.

There are several reasons for PreSoC to be open source.

Other users may have ideas that take PreSoC in a direction not conceived by this author. Anyone who calls himself or herself a software engineer can write better code than this. WE DO NOT WANT TO SUPPORT IT!

Conclusion

When it is necessary to write assembly code, a preprocessor can lessen the tedium with logical control constructs and add more logical instruction format. Both an executable program and source for PreSoC are provided to allow the reader to use it as is, or modify it for their own particular vision of development tools.

About the Author

Name: Dave Van Ess
Title: Principal Application Engineer
 Cypress MicroSystems

Background: BSEE from University of California, Berkeley.
 More than 27 Years experience in circuit, signal processing, digital, software, analog, and system design.

Holder of five U.S. Patents (plus three pending) for medical systems and signal processing. Holder of one U.S. Patent (plus one pending) for PSoC enhancements. Author of numerous Application Notes and technical articles.

Joined Cypress MicroSystems in 2000.

Contact: dww@cypress.com

Cypress Semiconductor
 2700 162nd Street SW, Building D
 Lynnwood, WA 98037
 Phone: 800.669.0557
 Fax: 425.787.4641

<http://www.cypress.com/>

Copyright © 2004-2005 Cypress Semiconductor. All rights reserved.
 PSoC™, Programmable System-on-Chip™, and PSoC Designer™ are trademarks of Cypress Semiconductor.
 All other trademarks or registered trademarks referenced herein are the property of their respective owners.
 The information contained herein is subject to change without notice. Made in the U.S.A.