



This document applies to version 1.0.0 of the Iron Spring PL/I preprocessor, the first non-beta version.

The Iron Spring PL/I Preprocessor is an implementation of a generalized processor developed to provide macro capabilities for the PL/I language, although it is also suitable for other applications. It has minimal dependence on the input format, and its macro language is a subset of PL/I.

ISPP can be downloaded from:

<http://www.iron-spring.com/download>

Report bugs to:

[matto.bugs@iron-spring.com/?Subject=ISPP](mailto:matto.bugs@iron-spring.com/?Subject=ISPP)

(success reports welcomed also)

## Compatibility

The Iron Spring Preprocessor is based on the preprocessor language described in the IBM PL/I for MVS & VM Language Reference, release 1.1 (SC26-3114-01). Some additional features are based on those found in later IBM PL/I compilers. Complete compatibility is not necessarily guaranteed. Some differences are flagged by ► ◀ in the text.

## Table of Contents

Compatibility.....	1
Input Format.....	3
Input text.....	3
Preprocessor Statements.....	3
Files.....	4
Preprocessor scan.....	4
Scope and type of identifiers.....	5
Preprocessor constants.....	6
Preprocessor expressions.....	7
Preprocessor Statements.....	8
DECLARE statement.....	10
ACTIVATE statement.....	11
DEACTIVATE statement.....	11
%assignment statement.....	12
PROCEDURE statement.....	13
Arguments.....	13
Arguments in preprocessor statements.....	13
Arguments in input text.....	13
STATEMENT keyword.....	14
RETURN statement.....	14
Flow of Control.....	15
DO statement.....	15
Simple DO.....	15
Iterative DO.....	15
END statement.....	16
Procedure.....	16
DO-group.....	16
ITERATE statement.....	16
LEAVE statement.....	16
IF statement.....	17
null statement.....	17
GOTO statement.....	17
NOTE statement.....	18
Preprocessor builtin functions.....	19
COMMENT builtin.....	19
COMPILETIME builtin.....	20
COUNTER builtin.....	20
DATE builtin.....	20
INDEX builtin.....	20
LENGTH builtin.....	20
PARMSET builtin.....	21
QUOTE builtin.....	21
SUBSTR builtin.....	21
SYSPARM builtin.....	22
SYSTEM builtin.....	22
TIME builtin.....	22
Running the Preprocessor.....	23

## Input Format

Preprocessor source is one or more files, containing a mix of *preprocessor statements* and *input text*. Preprocessor statements are indicated by a leading %, and have a syntax described below. (► To allow use of % characters in input text, code %% , which will cause one % to appear in the output and the following text to not be scanned as a preprocessor statement). ◀

### *Input text*

Input text is relatively free form, within margins specified by the user. Text contained in PL/I-style comments, delimited by /\*...\*/, and character strings, delimited by “...” or ‘...’ is not eligible for replacement. A quote character in input text must be represented by two quotes, or be enclosed in the alternate quote character (“” or “”). a % within a comment or character string will not be recognized as a preprocessor statement.

► The preprocessor attempts to place input text at the same column position as it appeared in the preprocessor source, to retain any formatting in the original source. ◀

### *Preprocessor Statements*

Preprocessor statements are headed by a single % character and terminated by a ‘;’. Statements are coded in a PL/I-like language and cause action to be taken when generating the output from the input text. Actions may involve replacing active identifiers with an arbitrary sequence of output characters, including, excluding, or repeating portions of text, and so on. A *Preprocessor Procedure* is a sequence of preprocessor statements which can be invoked as a function.

Preprocessor statements can be placed anywhere in the input text and are executed when the statement is encountered. Preprocessor procedures can be placed anywhere in the input and are executed only when called, whether from input text, a preprocessor statement, or another preprocessor procedure. Preprocessor procedures return a value to the point of invocation.

## Files

The name of the primary input file is passed to the preprocessor on the command-line. Conventionally this file may have the extension “.pli” or “.pp”. The preprocessor can optionally include a header file before the primary input. This file is identified by the switch -s <filename> on the ispp command.

An optional output *insource* file (“.ins”) lists all input as read, and any error messages and %NOTEs generated. If the insource listing is not requested, error and %NOTE messages are sent to SYSPRINT.

The preprocessor output is written to a file (“.dek”) suitable for input to the Iron Spring PL/I compiler or other applications.

[► Currently the preprocessor is a stand-alone program not integrated with the compiler. Output lines are limited to 100 characters, and the output will break on suitable boundaries if necessary. These are temporary restrictions.]◀

The input file need not be a valid PL/I program.

## Preprocessor scan

The Iron Spring PL/I Preprocessor reads the complete preprocessor input file. It identifies all preprocessor statements and procedures, and compiles them to an intermediate language. Following this it processes the input text in the file, executes all preprocessor statements as they are encountered, and copies the input text to the output, replacing any active preprocessor identifiers. Preprocessor statements may cause the scan to skip or repeat portions of input text. Preprocessor procedures are executed when they are called, not when they are defined.

The preprocessor scans the input text for *active identifiers* which may be replaced in the output.

## Scope and type of identifiers

A preprocessor identifier is up to 64 alphanumeric characters, the first must be alphabetic (A-Z, a-z, #\$\_@\_ )—case of identifiers is ignored.

Preprocessor identifiers declared outside of a preprocessor procedure have *global scope*; that is, they are defined and potentially active throughout the source program, including within procedures. Preprocessor identifiers declared within a procedure have *local scope*; they are defined only in the procedure in which they are declared. Local and global identifiers may have the same name, in which case the local declaration overrides the global. Declarations can be placed anywhere in the source or procedure and declarations don't have to precede use.

Preprocessor identifiers are defined by a %DECLARE statement, which specifies their type and optional replacement attributes.

Identifiers may be declared CHARACTER, FIXED, ENTRY, or BUILTIN. CHARACTER data is similar to PL/I VARYING CHARACTER variables with no fixed length. ► This version of the preprocessor limits CHARACTER data to 4096 bytes. ◀ FIXED data is similar to PL/I FIXED DECIMAL(5) data. BUILTIN declares one of a number of preprocessor builtin functions. ENTRY declares a preprocessor procedure, which may be defined elsewhere.

Identifiers have the scan attribute RESCAN or NORESCAN. RESCAN means that after replacement of the identifier the resulting text is again scanned for replacement. This is repeated until no identifiers in the scanned text are replaced. NORESCAN means that the resulting text is not scanned again after replacement. RESCAN is the default. Identifiers may be marked eligible or ineligible for replacement by the declaration, or by the %ACTIVATE and %DEACTIVATE statements.

**If RESCAN is in effect, there is no limit to the number of times the resulting text can be rescanned, as long as at least one identifier is replaced each time. It is possible that an infinite loop may occur. This is compatible with the behavior of IBM preprocessors.**

## Preprocessor constants

Preprocessor constants may be character strings up to 32K bytes, optionally signed fixed decimal integers of up to five digits, or bit constants up to 32 bits;

Character strings can be delimited by either single or double quote characters (' or "). The quotes must match. The quote character not used as a delimiter can appear in the string. If the delimiter character is included in the string it must be doubled (" or "").

Hexadecimal character strings consist of sequences of characters 0-9, A-F and a-f. They are enclosed in quotes and immediately followed by an 'X'. ► UTF-8 characters are allowed in hexadecimal character strings. For example, the hex value "C2AC"x will generate the character ñ (not) ◀

Decimal numeric constants consist of characters -,+, 0-9, or \_ (for readability). Hexadecimal numeric constants consist of sequences of hexadecimal characters plus '\_ ', enclosed in quotes, and immediately followed by 'XN' (to indicate signed) or 'XU' (to indicate unsigned). Note that numeric constants should not exceed the range -99999 to +99999.

Bit constants are strings of the characters 0, 1, and \_ enclosed in quotes immediately followed by B. Hexadecimal bit constants are sequences of hexadecimal characters in quotes followed by BX or B4. Octal bit constants are sequences of characters 0-7 and \_ in quotes followed by B3. ► No matter what the representation, bit constants cannot exceed 32 bits. ◀

### ***Constant examples***

'100000000'B

-5

1\_234

"444"Bx

"C2AC"x (UTF-8 NOT SIGN)

"7\_0f"xn

"This is a character string"



## Preprocessor Statements

Preprocessor statements, except within a preprocessor procedure, all begin with %<keyword>, except assignment statements and the NULL statement, and end with “;”. Statements may begin and end at any point in a line and may span lines.

Preprocessor statements may be labeled, with one or more statement labels. Statement labels are identifiers followed by ‘:’ preceding the statement. Preprocessor procedures must be labeled. Other statements may be labeled to allow them to be the target of a GOTO, LEAVE, or ITERATE. Labels are not shown in the syntax definitions below.

► Any text preceding the opening “%” and following the closing “;” is treated as input text. Comments within a preprocessor statement are ignored and not copied to output, therefore:

```
% /* This is a preprocessor comment */ ;
```

will appear as a comment in the *insource* listing, but not appear in the output text. ◀

► The following “%” statements are not processed by the preprocessor, but are copied unchanged to the output text: %PAGE; %SKIP[(n)]; %REPLACE;.

%INCLUDE; can optionally be executed by the preprocessor, and the included text preprocessed, or it can be copied to the output text unchanged. Currently this is all-or-nothing, and not on an individual file basis. ◀

**PREPROCESSOR STATEMENTS**

%DECLARE (%DCL)

%ACTIVATE (%ACT)

%DEACTIVATE (%DEACT)

%assignment-statement

%PROCEDURE (%PROC)

%DO

%END

▶ %ITERATE ◀

▶ %LEAVE ◀

%IF - %THEN - %ELSE

%GO TO (%GOTO)

%null-statement

%NOTE

***DECLARE statement***

The %DECLARE statement defines a preprocessor identifier, establishes it as a preprocessor variable, preprocessor procedure, or preprocessor builtin function, and activates it for replacement.

The syntax is:

```
<declaration> ::= "%" "DECLARE" <identifier_list>
  <attribute>
  [ ",", " declaration..." ";"
  (DECLARE may be abbreviated DCL)
<identifier_list> ::= identifier |
  "(" identifier [,identifier...] ")"
<attribute> ::= "CHARACTER" | "FIXED" | "ENTRY" |
  "BUILTIN"
  (CHARACTER may be abbreviated CHAR)
```

**ACTIVATE statement**

The %ACTIVATE statement marks an identifier eligible for replacement, and optionally changes its RESCAN status.

```
<Activate_statement> ::= "%" "ACTIVATE" <identifier>  
    [ "SCAN" | "RESCAN" ]  
    [ ", " <identifier> [ "SCAN" | "RESCAN" ] ... ] ";"  
    (ACTIVATE may be abbreviated ACT)
```

If SCAN or RESCAN is omitted, the default is RESCAN.

The %ACTIVATE statement takes effect when executed, and remains in effect until the end of the program, or until it is canceled by a %DEACTIVATE statement for the same identifier.

A %ACTIVATE statement for an active identifier has no effect, except possibly to change its scanning status

**DEACTIVATE statement**

The %DEACTIVATE statement marks an identifier ineligible for replacement.

```
<Deactivate_statement> ::= "%" "DEACTIVATE" <identifier>  
    [ ", " <identifier> ... ]  
    (DEACTIVATE may be abbreviated DEACT) ;
```

The %DEACTIVATE statement takes effect when executed, and remains in effect until the end of the program, or until it is canceled by an %ACTIVATE statement for the same identifier.

***%assignment statement***

The assignment-statement assigns a value to a preprocessor variable.

```
<assignment_statement> ::= <identifier> "=" <expression>  
                           ";"
```

<identifier> is any preprocessor variable except a statement label.

The preprocessor expression <expression> is evaluated (see "Preprocessor expressions" above), the result converted to the type of <identifier>, if necessary, and assigned to the value of <identifier>.

[BIT values are converted to FIXED, 0=FALSE, 1=TRUE.]

***PROCEDURE statement***

Preprocessor procedures are functions which can be called from other preprocessor statements, or through replacement of input text. A preprocessor procedure begins with a %label: PROCEDURE statement and ends with [%]END. At least one label is required on the PROCEDURE statement. A preprocessor procedure must return a value, which may be FIXED or CHARACTER. The syntax of the PROCEDURE statement is:

```
<Procedure_statement> ::= "%" <label> ":" [ <label> ":"...]
    "PROCEDURE"
    [ <parameter_list> ] [ "STATEMENT" ]
    "RETURNS" "(" [ "CHARACTER" | "FIXED" ] ")" ";"
    (PROCEDURE may be abbreviated PROC)

<parameter_list> ::= "(" <parameter>
    [ ",", <parameter> ... ] ")"
```

Preprocessor procedures cannot be nested.

**Arguments**

Arguments to preprocessor procedures are handled differently, depending on whether the procedure is called from a preprocessor statement or from input text. In all cases the number of arguments and parameters need not match. Missing arguments are passed as the null character string or zero. Extra arguments are ignored.

**Arguments in preprocessor statements**

When a preprocessor procedure is called from another preprocessor procedure or statement, each argument can be an identifier, a FIXED or CHARACTER constant, or an expression. An identifier argument will be passed by reference if its type matches the corresponding parameter. In all other cases a dummy argument will be created.

**Arguments in input text**

When a preprocessor procedure is called from input text each argument is a character string, delimited by a "," or a closing ")". Blanks in arguments are retained. Each argument is scanned for replacement and assigned to the corresponding parameter.

**STATEMENT keyword**

The STATEMENT keyword indicates that the procedure can be invoked from input text as if it were a PL/I statement. Arguments can be assigned in the normal fashion, or by name, or a mixture. For example a procedure declared as:

```
P: PROCEDURE(one, two, three) STATEMENT... ;
```

might be called as:

```
P ONE(a) TWO(b) THREE(c);
```

or:

```
P(a, c) TWO(b);
```

Which result in the same call.

Returned values of STATEMENT procedures replace all text between the procedure name and the closing ‘;’, inclusive.

**RETURNS keyword**

The RETURNS keyword is required, and indicates what type of value the procedure returns, FIXED or CHARACTER.

***RETURN statement***

A RETURN-statement exits a preprocessor procedure, and specifies the value to be returned to the point of invocation. The syntax is:

```
<Return_statement> ::= "RETURN" "(" <expression> ")" ";"
```

<expression> specifies the value to be returned. At least one RETURN-statement must be present in a preprocessor procedure. <expression> will be evaluated and, if necessary, converted to the RETURNS type for the procedure.

## ***Flow of Control***

The %DO, %END, %LEAVE, %ITERATE, %IF, %THEN, %ELSE, null, and %GOTO statements can be used to affect the sequence of processing.

### **DO statement**

The %DO statement, together with the %END statement, delimits a preprocessor DO-group, and optionally specifies repetition. The DO-group can contain input text, listing control statements, and preprocessor statements. A DO can either be a simple DO or an iterative DO, specifying repetition.

```
<Do_statement> ::= <simple_do> | <iterative_do>
```

#### **Simple DO**

```
<simple_do> ::= "% DO";
```

#### **Iterative DO**

```
<iterative_do> ::= "% DO <identifier> =" <expression>  
[ "TO" <expression> [ "BY" <expression> ] |  
"BY" <expression> [ "TO" <expression> ] ] ";"
```

A DO-group can be used anywhere a single statement may appear, for example as the *preprocessor unit* in a THEN or ELSE.

## END statement

A %END statement is used to terminate a preprocessor procedure or DO-group.

```
<end_statement> ::= "%" "END" [ <identifier> ] ";"
```

If present, <identifier> must be a statement label of a DO-statement or PROCEDURE-statement. The optional <identifier> indicates which procedure or DO-group is being ended, and one END can close multiple blocks.

### Procedure

An END statement for a preprocessor procedure marks the end of the procedure. A leading "%" is optional.

### DO-group

A %END-statement for a preprocessor DO-group marks the end of the group. For a simple DO, the group is exited. For an iterative DO, control returns to the group head to determine if further repetitions are required.

## ITERATE statement

- ▶ The %ITERATE statement transfers control to the %END statement of its containing %DO group, or of the named %DO group. The next iteration, if any, is started.

```
<iterate_statement> ::= "%" "ITERATE" [ <identifier> ]  
";" ◀
```

## LEAVE statement

▶ The %LEAVE statement exits its containing DO-group, or the group identified by <identifier>. A LEAVE-statement must be contained in a DO-group. The syntax is:

```
<leave_statement> ::= "%" "LEAVE" [ <identifier> ] ";"
```

<identifier> must be a label of a containing DO-statement. If <identifier> is omitted, LEAVE terminates the immediately containing group. ◀

## IF statement

A %IF statement selects one of two paths of execution, depending on the value of an expression.

```
<if_statement> ::= "%" "IF" <expression>  
                "%" "THEN" <preprocessor_unit_1>  
                [ "%" "ELSE" <preprocessor_unit_2> ] ";"
```

<preprocessor\_unit\_1> and <preprocessor\_unit\_2> can be any single preprocessor statement, or a preprocessor Do-group, which may contain any preprocessor statements or input text.

<expression> is any expression which can evaluate to the equivalent of BIT(1), where all bits zero = '0'B (false), and anything else = '1'B (true).

If the <expression> is true, <preprocessor\_unit\_1> is executed, and execution continues with the statement following the If-statement.

If the <expression> is false, <preprocessor\_unit\_2 (if present) is executed. If the expression is false and <preprocessor\_unit\_2> is not present, execution continues following the IF-statement.

## null statement

The null-statement has no effect; The syntax is:

```
<null_statement> ::= "%" ";"
```

The null statement may fill in for, for example, <preprocessor\_unit\_1> in an If-statement.

## GOTO statement

The %GOTO-statement causes the preprocessor to continue execution at another point in the preprocessor input file.

```
<goto_statement> ::= "%" "GOTO" <identifier> ";"  
                ("GO TO" is an alternative for "GOTO")
```

<identifier> must be a label of a preprocessor statement. This statement must not be contained in a different procedure or DO-group than the GOTO.

**NOTE statement**

THE NOTE statement generates a preprocessor message on the INSOURCE listing.

```
<note_statement> ::= "%" "NOTE"  
    "(" <message> [ ", " <severity> ")" ";"
```

<message> is a preprocessor expression that resolves to a CHARACTER value to be displayed as the text of the message.<severity> is an optional preprocessor expression that resolves to a FIXED value indicating the error level assigned to this message.

The <severity> should be 0, 4, 8, 12, or 16. Otherwise the results are undefined. If the <severity> is omitted, zero is the default. Messages with a <severity> of 16 cause immediate termination of preprocessing.

The following statement:

```
if -parmset(three)  
then note('Argument "three" missing',4);
```

Produces the following preprocessor message:

```
34 *** NOTE          4, Argument "three" missing
```

## Preprocessor builtin functions

The preprocessor has a number of builtin functions.

Preprocessor builtin functions
COMMENT
COMPILETIME
COUNTER
▶ DATE ◀
INDEX
LENGTH
PARAMSET
SYSPARM
QUOTE
SUBSTR
▶ TIME ◀

### ***COMMENT builtin***

The COMMENT builtin returns its argument as a CHARACTER value wrapped in `/* ... */`.

Examples:

```
COMMENT( This is a comment );
```

returns:

```
/* This is a comment */;
```

and:

```
%a = COMMENT("I'm a comment, too")
```

sets the preprocessor identifier "a" to the value:

```
/*I'm a comment, too*/
```

**COMPILETIME builtin**

The COMPILETIME builtin returns the date and time the preprocessor began execution as a CHARACTER value of length 18, as “dd MMM yy hh.mm.ss”

Example:

```
x = COMPILETIME;
```

returns the value:

```
x = 18 NOV 20 22.00.51;
```

(note that the string is not quoted and should be wrapped by the COMMENT or QUOTE builtins.

**COUNTER builtin**

The COUNTER builtin returns a FIXED value that begins at 00001 and increments by one each time it is called.

**DATE builtin**

► The DATE builtin returns a FIXED the current date when it is called, as a CHARACTER value of length 8, as “YYYYMMDD”. (This is a VAX PL/I function) ◀

**INDEX builtin**

The INDEX(x,y) builtin returns a FIXED value of the position in string “x” of a substring “y”. If “y” does not occur in “x”, or either “x” or “y” has zero length, zero is returned

**LENGTH builtin**

The LENGTH builtin returns the length of the character value of its argument. FIXED arguments return 5.

***PARMSET builtin***

The PARMSET builtin can be used only inside a preprocessor procedure to indicate whether or not an argument has been supplied. It returns '1'b if the named argument is present, and '0'b if not. For example, in a procedure headed:

```
%foo: procedure(bar, baz) returns(character);
```

PARMSET(baz) returns '1'b if the procedure is called with

```
%qux = foo('1', '2');
```

and '0'b if called with

```
%qux = foo('1'); or %qux = foo('1', );
```

► The current version of PARMSET checks only the parameters of the immediately containing procedure, and not of any invoking procedures. ◀

***QUOTE builtin***

The QUOTE builtin returns its argument as a CHARACTER value wrapped in single quotes. If the argument contains single quotes they are each converted to two single quotes.

Examples:

```
QUOTE( This is a quote );
```

returns:

```
' This is a quote ';
```

and:

```
%a = QUOTE("I'm a quote, too")
```

sets the preprocessor identifier "a" to the value:

***SUBSTR builtin***

The SUBSTR(x,y,[z]) builtin returns a portion of the string value of argument "x", beginning at position "y". The result continues through the end of "x", or for "z" characters.

***SYSPARM builtin***

The SYSPARM builtin returns a string value which was passed to the preprocessor in the command-line argument -p. Contents and interpretation of the SYSPARM value are left to the user.

***SYSTEM builtin***

► The SYSTEM builtin returns a string which contains a value indicating the operating system on which the preprocessor is running. This string contains the OS name, possibly followed by additional information which is currently undefined. Currently the returned string begins with the value “Linux”. ◀

***TIME builtin***

► The TIME builtin returns the current time when it is called, as a CHARACTER value of length 8, as “HHMMSSTT”. (This is a VAX PL/I function) ◀

## Running the Preprocessor

The command to run the preprocessor is:

```
ispp -li "-cn(^¬)" "-co(|)"  
      "-m(m1[,m2[,m3[,m4]])"  
      -i include_dir -I  
      -ed  
      -p "sysparm string"  
      -s startup file  
      file1.pli -o file2.dek
```

- All arguments are optional except the input file (file1). Arguments containing parentheses must be in quotes to keep the shell happy.
- ispp is the preprocessor name, this should be the path to the appropriate directory.
- file1.pli is the name of the input source file. The name format is arbitrary
- file2.dek is the name of the output file. The name format is arbitrary.
- -li is an option telling the preprocessor to produce an “insource” listing. If this option is not specified a listing file containing only preprocessor messages will be generated.
- -cn specifies alternate characters to be used for the logical NOT (¬) character. The default, from code page 850, is ‘AA’x, the Unicode Latin-15 value is ‘AC’x. Up to four other characters such as the caret (^-as shown) or exclamation mark (“bang” !) can be specified.
- -co specifies alternate characters to be used for OR. The default is ‘|’, ‘7C’x.
- -m specifies the margins for the input and output files. m1 and m2 are the input margins, m3 and m4 are output. If m is omitted the defaults for both are (1,100). Characters outside the range m1-m2 are ignored on input. The output is placed within the margins m3-m4.
- -I directs the preprocessor to process include files. Without -I, all %INCLUDE statements are passed thru to the compiler with no preprocessing. When -I is specified all included text will be preprocessed and embedded in the generated .dek file. This is a global option and is not currently specifiable on a file-by-file basis
- -ed instructs the preprocessor to display error and warning messages on

stderr in addition to printing them on the insource listing.

- -i supplies the directory for include files, defaults to “.”. -i may be coded multiple times to specify multiple directories.
- -p specifies a user-defined string to be passed to the preprocessor program through the builtin function SYSPARM. The format of this string is arbitrary, and the interpretation is left to the user program. The string is passed as-is, minus enclosing quotes, to the user program.
- -s specifies a file which will be prepended to the preprocessor source before scanning. This file may contain anything, but is especially intended to contain preprocessor macros which apply to all files in a project, and would otherwise have to be manually inserted into every program.
- -V requests the preprocessor to display the current version and exit.

No other options are valid at this time.

The directory “samp” contains the sample program as run by both IBM VisualAge PL/I for windows and ISPP. The source is the same, win.LST and win.DEK are the windows versions.

ISPP can be downloaded from:

<http://www.Iron-Spring.com/download/>