User's Guide for the ICC Source Code Formatter Version 2.1.0 Document Version 2.3.1 UNIX and Windows-NT Systems

January 26, 2001

Irvine Compiler Corporation
15707 Rockfield Blvd, Suite 305
Irvine, CA 92618

(949) 609-4650	Web:	www.irvine.com
FAX: (949) 609-4656	Sales:	sales@irvine.com
	Technical support:	support@irvine.com

This document was prepared with LaTeX.

1 General Description

ICC's source code formatter, `iccfmt`, formats the Ada source files specified on the command line in a manner similar to that used throughout the Ada Language Reference Manual (LRM). An output file is created with the same name as the input file except with the extension changed to `.fmt`. The format may be changed by using qualifiers on the command line or in an optional startup option file. The startup option file is placed in the user's home directory and named `.iccfmt`.

The startup option file can contain any options that may be specified on the command line. Ada-style comments are also supported. By using the startup option file, a user can automatically format files to standardized coding styles or to his/her own personal style.

Error recovery is minimal. If any errors should occur while parsing the input source file, *no* `.fmt` file is generated and at least one error message indicating the filename, line number of the first detected error, and type of the error is printed to standard output. Additional error checking may or may not be performed depending on the type of error(s) discovered.

2 Invocation

FORMAT

`iccfmt` [*qualifier(s)*] [*Ada source file(s)*]

The *Ada source file(s)* may be specified with or without the `.ada` extension.

QUALIFIERS

Least-matching on options is supported; hence, only enough characters to uniquely identify the option need to be typed.

`-ada83`—Specifies that the source file contains Ada83 syntax (i.e., the six Ada95 keywords may be used as identifiers). The default is `-ada95`.

`-ada95`—Specifies that the source file contains Ada95 syntax. The default is `-ada95`.

`-help`—Displays a summary of the `iccfmt` invocation, qualifiers, and qualifier default values.

`-output`

Format:

-output=<*name*>

Specifies the output file name. By default, the output is written to a file in the current working directory with same name and an extension of `.fmt`. This qualifier may not be specified when formatting multiple Ada source files.

`-quiet`—Executes quietly.

Page Formatting Options

`-comment_column`

Format:

-comment_column=<*number*>

Defines the column position for comments. The default is column 40. `iccfmt` attempts to place all comments beginning at this column. If a comment is too wide to fit the page (as defined by the `-page_width` qualifier), then it is placed at the current indentation column. If a comment appears at the end of a line ending with a semicolon, the comment is placed on that line if there is room. Otherwise, it is placed on the following line. If there are several comments in a group, this applies to the whole group, not just the individual comments. For example, if a comment group consists of five lines, one of which is too long to be placed at the specified column, then the entire group will be placed at the current indentation column. Comment lines too long for the specified page width are *not* split.

Comment placement is not perfect. Comments appearing on the last line of a block construct (`if`/`then`, loops, etc.) will appear after the end of the construct. Comments just before a `begin` (or the first statement of a block construct) will be placed after the statement. Blank lines between comments are ignored; the comments will be treated as a single group.

It is recommended that comments not appear in the middle of any statement or declaration. This includes using comments on each line to identify each item of a parameter list in a subprogram declaration.

`-context_column`

Format:

-context_column=<*number*>

Defines the column position for `with`/`use` clauses. The default is column 1. Both the `with` clause and the associated `use` clause are placed on the same line if there is enough room.

`-dotdot`—Pads both sides of "`..`" constructs with spaces. For example, if `-dotdot` is specified on the command line, the declaration

```
const_str : constant STRING(1..7) := "ABCDEFG";
```

will be printed as

```
const_str : constant STRING(1 .. 7) := "ABCDEFG";
```

`-pad_aggregate`—Pads both sides of a parenthesized aggregate with spaces.

`-page_width`

Format:

-page_width=<*number*>

Defines the page width. The default is 79.

`-parameter_align`—Enables parameter mode alignment. The default is not to align parameter modes. When enabled, `iccfmt` will attempt to place all parameter mode keywords (e.g., `in`, `out`, `in out`) in the same column within a subprogram declaration.

`-strip_comments`—Strips the source code of all comments. The default is to include comments within the generated source code. Comment placement is not perfect. It is recommended that comments not appear in the middle of any statement or declaration. This includes using comments on each line to identify each item of a parameter list in a subprogram declaration.

Indentation Options

`-indentation`

Format:

-indentation=<*number*>

Defines the number of columns to indent for constructs such as the `if` statement and loops. The default indentation is 3. The indentation level for each construct (`if`, `for`, `loop`, `pragma`, and `while` blocks) can also be set individually. See `-if_indent`, `-for_indent`, `-while_indent`, `-loop_indent`, and `-pragma_indent`.

`-if_indent`

Format:

-if_indent=<*number*>

Defines the indentation level for the `if` statement. If this qualifier is not specified, the indentation level is that specified by the `-indentation` qualifier.

`-for_indent`

Format:

-for_indent=<*number*>

Defines the indentation level for `for` loops. If this qualifier is not specified, the indentation level is that specified by the `-indentation` qualifier.

`-loop_indent`

Format:

-loop_indent=<*number*>

Defines the indentation level for `loop` blocks. If this qualifier is not specified, the indentation level is that specified by the `-indentation` qualifier.

`-pragma_indent`

Format:

-pragma_indent=<*number*>

Defines the indentation level for `pragma` statements. In the case of `pragma if` *blocks*, all the statements inside the block will also be indented. This, however, will cause unpredictable results in cases where the statements are not well nested (see Section 5). If this qualifier is not specified, the indentation level is that specified by the `-indentation` qualifier.

`-while_indent`

Format:

-while_indent=<*number*>

Defines the indentation level for `while` loops. If this qualifier is not specified, the indentation level is that specified by the `-indentation` qualifier.

Alphabetic Casing Options

For these qualifiers, the valid values for *casing* are:

`upper` `lower` `mixed` `untouched`

`-declaration_case`

Format:

-declaration_case=<*casing*>

Defines the casing for declaration names. The default is to leave the casing `untouched`.

`-keyword_case`

Format:

-keyword_case=<*casing*>

Defines the casing for keywords (e.g., `procedure`, `begin`, `if`, etc.). The default is `lower` case.

`-object_case`

Format:

-object=<*casing*>

Defines the casing for identifiers when they are referenced. The default is to leave the casing `untouched`.

`-unit_case`

Format:

-unit=<*casing*>

Defines the casing for unit names in `with` and `use` clauses. The default is to leave the casing `untouched`.

Formatting Options

`-aggregate_format`

Format:

-aggregate= `single` | `paragraph`

Defines the format for simple lists; for example, an assignment to an array. `single` results in each element assignment being printed on a separate line. `paragraph` results in the entire assignment being printed in the form of a paragraph and is the default.

`-enumeration_format`

Format:

-enumeration=`single` | `multi` | `paragraph`

Defines the format for enumeration types. `single` results in each item of the enumeration appearing on a separate line `paragraph` results in the entire enumeration being printed as a single paragraph (i.e., as though it were text). `multi` results in the enumeration being printed in a columnar format. The default is `multi`.

3 Ada95 Features

`iccfmt` has been enhanced to support the full Ada95 syntax.

Note: ICC makes no commitment to provide a full Ada95 compiler as a free upgrade.

4 Examples

In each of the examples, the words in *slanted* font are entered by the user, and the text in `typewriter` font displays the normal output from `iccfmt`.

Assume the file `simple.ada` (following) exists:

```
-- A simple program for testing the ICC pretty printer.

with text_io;
use text_io;

procedure simple is

package int_io is new integer_io(integer);
use int_io;

begin
int_io.default_width := 0;

for i in 1..10 loop
put(i);
put(' ');
end loop;
new_line;
end;
```

1. This example shows the default formating of `iccfmt`.

```
-- A simple program for testing the ICC pretty printer.

with text_io;
use  text_io;

procedure simple is
   package int_io is new integer_io(integer);
   use int_io;
begin
   int_io.default_width := 0;

   for i in 1..10 loop
      put(i);
      put(' ');
   end loop;

   new_line;
end simple;
```

2. This example shows the same file formatted with different qualifiers.

   % iccfmt -unit=upper -dotdot -for=5 simple

   The resulting file `simple.fmt`:

```
-- A simple program for testing the ICC pretty printer.

with TEXT_IO;
use  TEXT_IO;

procedure SIMPLE is
   package int_io is new integer_io(integer);
   use int_io;
begin
   int_io.default_width := 0;

   for i in 1 .. 10 loop
        put(i);
        put(' ');
   end loop;

   new_line;
end simple;
```

3. This example shows the output when the source file contains an error(s). Using the original `simple.ada`, assume that you have inadvertently omitted the semicolon (`;`) after `use text_io`.

   % iccfmt simple.ada

```
ICC Ada Source Code Formatter [vx.x.x MMM DD, YYYY]
"simple.ada", 2: Error: Syntax error before token "with".
"simple.fmt": Info: Deleting file because of errors.
```

5 Caveats

In `pragma if` *blocks*, all the statements inside the block will be indented. This will cause unpredictable results in cases where the statements are not well nested.

```
...
for i in m..n loop
   x = 10;
   pragma if ...
      ...
      end loop;
   pragma else ...
      ...
      end loop;
   pragma end if ...
-- the pragma if-else-endif block determines
-- the end of the for loop
...
```