Writing Template Files for Software AG IDL Compiler

An IDL template file contains the rules that the Software AG IDL Compiler uses - together with the IDL file - to generate interface objects, skeletons and wrappers for a programming language. The Developer’s Kit provides several templates for various programming languages.

Warning:
The information in this section is intended for users who wish to write their own template files. Do not change the delivered template files.

This document provides an introduction on how to write template files. The syntax for IDL Template Files in a formal notation is presented in the document *Grammar for IDL Template Files*.

This chapter covers the following topics:

- Coding Tempate Files
- Using Output Statements in the Template File
- Inserting Comments in the Template File
- Using Verbatim Mode
- Using Options
- Specifying the Name of the Output File
- Redirecting the Output to Standard Out
- Using Template #if Preprocessing Statements
- Using Template #include Preprocessing Statements
- Using Template #trace Statement

Coding Tempate Files

It is the combination of control and output statements (see *`control_statement`*, *`output_statement`* and *Using Output Statements in the Template File*) that provides the full definition of the target programming-language source code.

Usually a template file has *`definition-statement`* grouped together at the beginning; these are followed by loop_statements:

```
; type definitions
%using A "char %name%index"
....
; loop libraries
%library
```

```
{
         ....
         ; loop programs
         %program
         {
                 ....
                 ; loop parameters
                 %name
                 {
                         ....
                 }
          }
}
```

Using Output Statements in the Template File

Output statements (see *output_statement*) provide the actual templates of the target-language source code. Output statements are text strings enclosed in double quotes.

These text strings may contain *output_substitution_sequence*, *output_formatting_sequence*, *output_escape_sequence* and *output_of_variable*.

Substitution Sequences

Substitution sequences are identified by a preceding % and are substituted by their actual contents during generation.

Example: `"This is a text string. The library name is %library. \n"`

The IDL Compiler provides substitution sequences for the current library name, program name, parameter name, type, etc. Some substitution sequences can only be accessed in their corresponding loop statement. For example a `%program` substitution sequence is only valid within an active program loop (see *loop_over_programs*). See *output_substitution_sequence* for a list of valid substitution sequences and description.

Formatting Sequences

Formatting sequences are identified by a preceding \

Example: `"\n is a Formatting sequence"`

See *output_formatting_sequence* for a list of valid formatting sequences

Escape Sequences

Escape sequences are identified by a preceding \\

The escape character is used to change the meaning of special characters (&, ? and # etc.) back to their normal meaning. Special characters are used to access variables (see *output_of_variable*).

Example:`"\\&"`.

See also *`output_escape_sequence`* and *Using Verbatim Mode*.

Variables

The output of variables is forced when the special characters &, ? or # occur before the variable name (see *`variable_name`*) in output statements (see *Using Output Statements in the Template File*)

Example: `"?A is the output of a variable"`

See also *`output_of_variable`*

Generating Programming-language-specific Type Definitions

The substitution sequence `%type` is usually used in a parameter loop (loop_over_parameters) to generate programming-language-specific type definitions. Before the parameter loop all IDL data types (with *`definition-of-base-type-template`* statements) and the dimension information (with *`definition-of-index-template`* statements) must be specified.

Example

IDL data type I2 can be specified as follows in a C program:

```
%using %index "" "[%1_index]" "[%1_index][%2_index]"
"[%1_index][%2_index][%3_index]"
%using I2 "short %name%index;"
```

`%using %index` is the control_statement for the dimension information (*`definition-of-index-template`*). How the following strings are used depends on the dimension of the parameter. The first empty string is used for scalar parameters, the second for 1-dimensional parameters, the third for 2-dimensional parameters and the fourth for 3-dimensional parameters.

`%using I2` is the control_statement for the IDL data type I2 (see *`definition-of-base-type-template`*), "short %name%index" is the output_statement for this data type. `%name` and `%index` are substitution sequences. `%name` will be replaced by the variable name and `%index` will be replaced by any dimension information.

If an input IDL file contained the following parameter definitions:

```
1 Field-1   (I2)
1 Field-2   (I2/1:8)
1 Field-3   (I2/1:4,4:7)
```

then, based on the above template specifications, all references to the `%type` substitution sequences in any `output_statement` would be replaced by

```
short Field_1;
short Field_2[8];
short Field_3[4][4];
```

Generating Programming-language-specific Names

Special characters within some substitution sequences e.g. `%library`, `%program` and `%name` can be changed during generation to provide valid names for the target programming language. The IDL Compiler supports generation of names for the programming languages C, C# and COBOL (see `output_control_lower_upper` and `output_control_sanitize`).

Target Programming Language	Class Names	Function Names	Variable or Parameter Names
C	not applicable	`%UpperCase-` `%LowerCase+` `%Sanitize+`	`%UpperCase-` `%LowerCase+` `%Sanitize+`
C#	`%UpperCase-` `%LowerCase-` `%SanitizePascalCased+`		`%UpperCase-` `%LowerCase-` `%SanitizeCamelCased+`
COBOL	not applicable		`%SanitizeCobol+` `%UpperCase+` `%LowerCase-`

The default programming language when you do not code any `output_control_lower_upper` and `output_control_sanitize` statements in your template is C.

Inserting Comments in the Template File

Comments

- are identified by a ";" in a line and
- are terminated by the end of line.

For example:

```
; This is a comment
               ; So is this.
"output text followed by a comment" ; here is the comment
```

Whereas this is an output statement:

```
"an output text with a semicolon ;"
```

Using Verbatim Mode

If your output is going to contain many special characters, you may enter verbatim mode. Then all characters are written to the output as typed. The only sequences recognized in this mode are the escape sequences (see `output_escape_sequence`).

To enter verbatim mode

- Use the command `%verbose+` (see `output_control_verbose`).

  Example: In verbatim mode, you enter "&" to insert an ampersand.

Using Options

The IDL Compiler supports options within templates.

You can pass them with the parameter `-D` to the IDL Compiler (see *Starting the IDL Compiler*).

Options

- can be used in output statements (see *`output_of_variable`*)
- can be used in logical condition criteria (see *`compare_strings`*) in %if (see *`if_statement`*) and %while (see *`loop_of_while`*) statements
- are case-sensitive, i.e. hugo and HUGO are distinct options

Specifying the Name of the Output File

The name of the output file is controlled by the %file statement.

If the `%Format` substitution sequence in a file (`%file`) statement is used, the base name can be provided with the IDL Compiler parameter `-F` (see *Starting the IDL Compiler*).

If no base name is provided with the `-F` IDL Compiler parameter, the base name of the IDL file without path and extension is used as the default of the substitution sequence `%Format`.

See the following excerpt from a template file:

```
%file "C%F.c"
```

When the IDL Compiler is called with

-
  ```
  erxidl -t client.tpl .. \MyDirectory\example.idl
  ```

  an output file with the default base name *Cexample.c* of the IDL file is created

-
  ```
  erxidl -t client.tpl -Ftest example.idl
  ```

  an output file with the name *Ctest.c* is created

See also the IDL Compiler option `-o` (see *Starting the IDL Compiler*) on how to specify the directory for the output file.

Redirecting the Output to Standard Out

The output can be redirected to standard out with an environment variable (see *Using Options*), e.g. `NOOPEN`. This is optional.

See the following excerpt from a template file:

```
 %if "$(NOOPEN)" <> "1" %file "C%library.c" ;
```

When the IDL Compiler is called with

-
  ```
  erxidl -t client.tpl -D NOOPEN=1 example.idl
  ```

  the output is redirected to standard out

-
  ```
  erxidl -t client.tpl example.idl
  ```

  the output is directed to the file *Cexample.c* as specified in the template.

Using Template #if Preprocessing Statements

The IDL Compiler supports `#ifdef`, `#elif`, `#else` and `#endif` preprocessing statements similar to the C compiler preprocessor.

You can use preprocessor variables with the option -D (see *Starting the IDL Compiler*).

Additional rules for `#if` preprocessing statements are:

- If `#elif` is used, it must follow `#ifdef`.
- If `#else` is used it must follow either `#ifdef` of `#elif`.
- `#endif` must always close the `#ifdef` statement.
- Embedded preprocessor statements or logical concatenation of definitions are not allowed.

See the following excerpt from a template file:

```
#ifdef Definition_1
              "/* codes of -PDefinition_1 */\n"
              %name
              {
              :
              }
       #elif Definition_2
              "/* codes of -PDefinition_2 */\n"
              %name
              {
              :
              }
       #else
              "/* codes of neither Definition_1 nor Definition_2 */\n"
              %name
              {
              :
              }
       #endif
```

When the IDL Compiler is called with

-
  ```
  erxidl -t template_file -PDefinition_1
  ```

  the template statements "`/* codes of -PDefinition_1 */\n`" between the `#ifdef` and first `#elif` statement are interpreted.

-
  ```
  erxidl -t template_file -PDefinition_2
  ```

  the template statements "`/* codes of -PDefinition_2 */\n`" between the first `#elif` and second `#elif` statement are interpreted.

-

```
erxidl -t template_file
```

the template statements "`/* codes of neither Definition_1 nor Definition_2 */\n`" between the `#else` and `#endif` statement are interpreted.

See the following preprocessing statements with invalid syntax:

```
..
#ifdef (MY_VERSION) ; brackets are not allowed
#endif
#ifdef MY_VERSION || HIS_VERSION ; logical OR is not allowed
..
#endif
#ifdef (MY_VERSION)
..
#ifdef (MY_NEW_VERSION) ; embedded #ifdef is not allowed
```

Using Template #include Preprocessing Statements

The IDL Compiler supports `#include` preprocessing statements similar to the C compiler preprocessor. All statements in the included template file are simply embedded.

To find included template files, use the IDL Compiler option `-I` and add a list of directories that form a search path (see *Starting the IDL Compiler*).

- First the IDL Compiler searches for templates in the directory of the initial template.
- When no template is found in the directory of the initial template, all directories specified with `-I` are searched in the order of occurrence

Additional rules for `#include` preprocessing statements are:

- A maximum of 32 templates can be included in a generation process.
- An included template file can include further template files.
- Recursive inclusion of template files is not permitted.
- All variables can be accessed in all included template files as well as in the starting (root) template.

The compiler searches for included templates.

See the following excerpt from a template file:

```
#include "template.tpl"
```

Using Template #trace Statement

The IDL Compiler supports the `#trace`*`tracelevel`* statement to enable and disable template tracing within a certain block of the template. The usage of *`tracelevel`* is the same as the command-line option "`-T`".

See also compiler option "`-T`" under *Starting the IDL Compiler* for trace level values.

Example

```
#trace 2   ; enable tracing on trace level 2
%compute i "0"
%while "&i" < "10"
{
  %compute i "&i + 1"
}
#trace 0   ; disable
```