11 Pragmas

Pragmas are directives to the compiler, used to control or influence the translation process but they do not change the meaning of the program text. There are two types of pragmas:

- language defined pragmas
- implementation defined pragmas

11.1 Language Defined Pragmas

Language defined pragmas are portable across implementations. They names of language defined pragmas are always all-uppercase words. There are four types of language defined pragmas:

- a pragma to verify source text encoding
- pragmas to control conditional compilation
- pragmas to emit console messages during compilation
- pragmas to control, influence or optimise code-generation

11.1.1 Source Text Encoding Control Pragma

The source text encoding control pragma is used in the absence of an identifying BOM in the source text to ensure that the current encoding of the source text matches the encoding used when it was last encoded. A verification is carried out by matching quoted character literals against their code points within an arbitrary list of selected samples within the pragma body. Samples should be selected to represent code points that can identify the encoding scheme used. If any of the samples do not match, a fatal compilation error will result and compilation will be aborted.

The encoding pragma controls whether or not characters outside of the range of the US-ASCII[1] character set are allowed within comments, quoted character and string literals. By default, when no BOM is present and no encoding control pragma is specified in the source text, the only characters allowed within comments and quoted literals are printable characters within the range of code points 020U (whitespace) to 07EU (tilde). Any other characters will then cause a compilation error.

If an encoding control pragma is present in the source text, it must occur before any other token. There can only be one encoding control pragma per source file. The maximum number of code point samples accepted within an encoding control pragma is implementation defined. Any samples in excess of the implementation defined maximum may be ignored in which case a compile time warning is emitted. The maximum should not be lower than 10.

EBNF:

```
encodingControlPragma :
    "<*" ENCODING "=" quotedStringLiteral ( ":" codePointSampleList )? "*>" ;
codePointSampleList :
    codePointSample ( "," codePointSample )* ;
codePointSample :
    quotedCharacterLiteral "=" characterCodeLiteral ;
```

Examples:

```
<* ENCODING = "ASCII" *> (* force 7-bit ASCII only *)
<* ENCODING = "UTF8" : "é" = 0E9, "©" = 0A9U, "€" = 20ACU *>
```

[1] The term US-ASCII refers to the 7-bit ISO646-US character set.

11.1.2 Conditional Compilation Pragmas

Conditional compilation pragmas are used to compile sections within the source text only if a certain condition is met. The condition must be a compile time expression.

EBNF:

```
conditionalCompilationPragma :
    "<*" ( IF | ELSIF ) inPragmaExpression | ELSE | ENDIF "*>" ;
```

Example:

```
TYPE Model = ( small, large, custom );
<* IF TSIZE(INTEGER) = 2 *>
CONST model = Model.small;
<* ELSIF TSIZE(INTEGER) = 4 *>
CONST model = Model.large;
<* ELSIF TSIZE(INTEGER) MOD 2 = 0 *>
CONST model = Model.custom;
<* ELSE *>
<* FATAL "unsupported type model." *>
<* ENDIF *>
```

11.1.3 Compile Time Console Message Pragmas

Compile time console message pragmas are used to emit console messages during compilation. There are four types of message pragmas:

- pragmas emitting informational messages
- pragmas emitting compilation warnings
- pragmas emitting compilation errors
- pragmas emitting fatal compilation errors

Pragmas emitting informational messages may also be used to write the current value of a pragma controlled compiler setting to the console. Informational messages and warnings do not cause the compilation to fail. Error messages cause the compilation to fail but continue. Fatal messages cause the compilation to fail and abort immediately.

EBNF:

```
compileTimeMessagePragma :
    "<*" INFO ( compileTimeMessage | ALIGN | implDefinedPragmaName ) |
         ( WARN | ERROR | FATAL ) compileTimeMessage "*>" ;
```

Examples:

```
<* INFO ALIGN *> (* prints current value of memory alignment to console *)
<* FATAL "unsupported target architecture." *>
```

11.1.4 Code Generation Pragmas

Code generation pragmas are used to control or influence code generation and optimisation. There are six code generation pragmas:

- a pragma to force specified memory alignment
- a pragma to force specified calling convention
- a pragma to cause the build system to invoke an external utility
- a pragma to suggest inlining a procedure
- a pragma to suggest not inlining a procedure
- a pragma to mark a variable as volatile

EBNF:

```
codeGenerationPragma :
    "<*" ( ALIGN "=" inPragmaExpression |
           FOREIGN ( "=" quotedStringLiteral )? |
           MAKE "=" quotedStringLiteral |
           INLINE | NOINLINE | VOLATILE ) "*>" ;
```

Examples:

```
TYPE Point = RECORD <* ALIGN = 8*TSIZE(CARDINAL) *> x, y : OCTET END;

<* FOREIGN = "C" *> DEFINITION MODULE stdio;

<* MAKE = "genhashes foo bar baz > Hashes.def" *>

<* INLINE *> PROCEDURE P;

<* NOINLINE *> PROCEDURE Q;

VAR <* VOLATILE *> signal : Signal;
```

11.2 Implementation Defined Pragmas

Implementation defined pragmas are compiler specific and non-portable. The names of implementation defined pragmas may only be lowercase or mixed case words. Implementation defined pragmas may have any of the following modifiers:

- a trailing plus to turn on an on/off compilation setting
- a trailing minus to turn off an on/off compilation setting
- a trailing equal sign followed by a constant expression to set the value of a compilation setting

EBNF:

```
implementationDefinedPragma :
    "<*" pragmaName ( "+" | "-" | "=" inPragmaExpression )? "*>" ;

pragmaName : Ident ;
```