XcodeML/C
Specification

Version 0.9E (March 5, 2009)

XcalableMP/Omni Compiler Project

March, 2009

Copyright ©2008-2017 Omni Compiler Project (RIKEN AICS), Permission to copy without fee all or part of this material is granted, provided the XcalableMP Specification Working Group copyright notice and the title of this document appear. Notice is given that copying is by permission of Omni Compiler Project (RIKEN AICS).

History

Version 0.9E: March 5, 2009

- Modified elements for referencing array elements.
- Modified the `subArrayRef` element.
- Added the `indexRange` element.

Contents

1 Introduction **iii**

2 `XcodeProgram` element **iii**
2.1 `name` element . . . iii
2.2 `value` element . . . iii

3 `typeTable` element **iv**
3.1 Data type names . . . iv
3.2 `basicType` element . . . v
3.3 `pointerType` element . . . v
3.4 `functionType` element . . . vi
3.5 `arrayType` element . . . vi
3.6 `structType` and `unionType` elements . . . vii
3.7 `enumType` element . . . vii
3.8 Optional attributes for data type definition elements . . . viii

4 Symbol list **viii**
4.1 `id` element . . . viii
4.2 `globalSymbols` element . . . ix
4.3 `symbols` element . . . ix

5 `globalDeclarations` element **x**
5.1 `functionDefinition` element . . . x
5.2 `params` element . . . x
5.3 `varDecl` element . . . x
5.4 `functionDecl` element . . . xi

6 Statement element **xi**
6.1 `exprStatement` element . . . xi
6.2 `compoundStatement` element . . . xi
6.3 `ifStatement` element . . . xi
6.4 `whileStatement` element . . . xii
6.5 `doStatement` element . . . xii
6.6 `forStatement` element . . . xii
6.7 `breakStatement` element . . . xii
6.8 `continueStatement` element . . . xii
6.9 `returnStatement` element . . . xii
6.10 `gotoStatement` element . . . xii
6.11 `statementLabel` element . . . xii
6.12 `switchStatement` element . . . xiii
6.13 `caseLabel` element . . . xiii
6.14 `gccRangedCaseLabel` element . . . xiii
6.15 `defaultLabel` element . . . xiii
6.16 `pragma` element . . . xiii
6.17 `text` element . . . xiii
6.18 Line number attribute . . . xiii

7 Expression element **xiii**
7.1 Constant element . . . xiv
7.2 Elements that reference variables . . . xiv
7.3 `pointerRef` element . . . xv
7.4 Elements for referencing array elements . . . xv
7.5 Element for referencing structure members . . . xv
7.6 `assignExpr` element . . . xv
7.7 Binary operation elements . . . xvi
7.8 Unary operation elements . . . xvii
7.9 `functionCall` element . . . xvii
7.10 `commaExpr` element . . . xvii
7.11 `postIncrExpr`, `postDecrExpr`, `preIncrExpr`, and `preDecrExpr` elements . . . xvii
7.12 `castExpr` element . . . xviii
7.13 `condExpr` element . . . xviii
7.14 `gccCompoundExpr` element . . . xviii

8 XcalableMP element **xviii**
8.1 `coArrayType` element . . . xviii
8.2 `coArrayRef` element . . . xix
8.3 `subArrayRef` element . . . xix

9 Other elements and attributes **xix**
9.1 `typeName` element . . . xix
9.2 `is_gccExtension` attribute . . . xix
9.3 `gccAsm`, `gccAsmDefinition`, and `gccAsmStatement` elements . . . xx
9.4 `gccAttributes` element . . . xxi
9.5 `builtin_op` element . . . xxiii
9.6 `is_gccSyntax` attribute . . . xxiv
9.7 `is_modified` attribute . . . xxiv

10 Code examples **xxiv**

List of Figures

List of Tables

1 Introduction

Xcode for the C programming language is intermediate code that can be used to reconstruct a C program. This specification describes the Xcode XML representation.

This intermediate code level has the following characteristics:

- Preserves information that can be used to reconstruct a C program,
- Can represent the type information of the C programming language,
- Has syntax elements necessary for a variety of transformations, and
- Has a human-readable format (XML).

C-front is used to convert a C program to an XcodeML file. A decompiler is used to convert from an XcodeML file to a C program. Xcode programs can be created and analyzed using an analysis program.

2 `XcodeProgram` element

Programs written in Xcode are constructed from external definitions, comprising a Type table and a global Id table. The top-level element in an Xcode file is the `XcodeProgram` element. The `XcodeProgram` element includes the following elements:

- `typeTable` element - information on data type used by the program,
- `globalSymbols` element - information on global variables used by the program, and
- `globalDeclarations` element - information about function and variable declarations.

The elements have the following information attributes: /newline

- `compiler-info` - C-to-C compiler information,
- `version` - C-to-C compiler version information,
- `time` - Date and time of compilation,
- `language` - source language information, and
- `source` - source information.

2.1 `name` element

This element is used to specify a name, such as a variable name or a type name. The name is a character string. The element attributes are type attributes. The attribute values are type identifiers.

2.2 `value` element

This element is used to specify initial values. This element has the following subelements.

- `expression` (0 - multiple items) - expression that specifies a value
- `value` - nested value. This corresponds to "`...`".

Example

The following expression sets the initial value of an int type to `1`.

C code

```
<value>
  <intConstant type="int">1</intConstant>
</value>
```

The following expression sets the initial value of an int type vector to `1,2`.

XcodeML

```
<value>
  <value>
    <intConstant type="int">1</intConstant>
    <intConstant type="int">2</intConstant>
  </value>
</value>
```

3 `typeTable` element

The `typeTable` element is used to define the data type information for the entire file. The element is one of the data type definition elements. Data type definition elements comprise the following elements:

- `pointerType` element,
- `functionType` element,
- `arrayType` element,
- `structType` element,
- `unionType` element,
- `enumType` element, and
- `basicType` element.

3.1 Data type names

Data in a program are differentiated using data type names. These names are one of the following two type names:

- Basic data type names, which can be further split into those that correspond to C language basic data types:

  'void', 'char', 'short', 'int' , 'long', 'long_long', 'unsigned_char', 'unsigned_short', 'unsigned', 'unsigned_long', 'unsigned_long_long', 'float', 'double', 'long_double', 'wchar_t', 'bool' ('_Bool type)

  that correspond to _Complex and _Imaginary types:

  'float_complex', 'double_complex', 'long_double_complex', 'float_imaginary', 'double_imaginary', 'long_double_imaginary'

that correspond to types built into the GCC:

'__builtin_va_arg'

- Derived data type names - arbitrary alphanumeric strings that are not part of the above mentioned basic data type names Derived data type names must be unique within a program.

3.2 basicType element

The basicType defines the basic data type for C and C99. The element has the following attributes:

- type and
- name

Example

C code

```
struct {int x; int y;} s;
struct s const * volatile p;
```

is converted to the following XcodeML

XcodeML

```
  <structType type="S0">
    ...
  </structType>
  <basicType type="B0" is_const="1" name="S0"/>
  <pointerType type="P0" is_volatile="1" ref="B0"/>
```

3.3 pointerType element

The pointerType element defines a pointer data type. The element has the following attributes:

- type - the derived data type name for the pointer type and
- ref - the data type name used for referencing the pointer type and data type.

The pointerType element does not possess any other elements.

Example

The following data type definition corresponds to "int *".

XcodeML

```
  <pointerType type="P0123" ref="int" />
```

3.4 functionType element

The functionType element defines a function data type.

- type - the derived data type name for the function type.
- return_type - name of the returned data type for the function type.
- is_inline - specifies whether the function type is an inline type, using 0 or 1 (false or true).

If there is a prototype declaration, it includes the param elements corresponding to the argument elements.

Example

For "double foo(int a, int b)", the following corresponds to the "foo" data type.

XcodeML

```
<functionType type="F0457" return_type="double">
<params>
    <name type="int">a</name>
    <name type="int">b</name>
  </params>
</fucntionType>
```

3.5 arrayType element

The arrayType element defines an array data type. The arrayType element has the following attributes:

- type - the derived data type name for the array type,
- element_type - specifies the identifier for the array element data type,
- array_size - specifies the size of the array (number of elements). Omitting array_size and its subelement arraySize corresponds to not specifying the size. Attributes for array_size and its subelement arraySize cannot be specified simultaneously.
- is_const, is_volatile, is_restrict, is_static - indicates whether these attributes specify each of the const, volatile, restrict, and static modifiers of the array size. The value for each is 0 or 1 (false or true).

The element has the following subelement:

- arraySize - Expression specifying the size of the array (number of elements). The element has an expression subelement.

  When the size cannot be specified using a number, specify a variable-length array. When an arrayType element has an arraySize subelement, the array_size attribute value is "*".

Example

In "`int a[10]`" the type_entry corresponding to "`a`" is as follows.

XcodeML
```
<arrayType type="A011" element_type="int" array_size="10"/>
```

3.6 `structType` and `unionType` elements

A struct (structure) data type is defined using the `structType` element. The structType element has the following attribute:

- `type` - the derived data type name for the array type.

A union data type is defined using the `unionType` element. The `unionType` and `structType` elements have the same attributes and elements. The `structType` and `unionType` elements have symbolic elements that contain member identifier information. When structure and union tag names exist, they are defined in a symbol table that corresponds to the scope. The member bit field is described in the `bit_field` attribute of the `id` element, or in its subelement's `bitField` tag. The `bitField` tag has an expression subelement. Furthermore, the `id` element, which possesses the `bitField` tag, has a `bit_field` attribute, whose value is "`*`".

Example

In "`struct int x; int y : 8; int z : sizeof(int); S;`" the `structType` element corresponding to `S` is as follows:

XcodeML
```
<stuctType type="S6e89">
  <symbols>
    <id type="int">
      <name>x</name>
    </id>
    <id type="int" bit_field="8">
      <name>y</name>
    </id>
    <id type="int" bit_field="*">
      <name>z</name>
      <bitField>
        <sizeOfExpr>
          <typeName ref="int"/>
        </sizeOfExpr>
      </bitField>
    </id>
   </symbols>
</structType>
```

3.7 `enumType` element

The `enumType` element defines the `enum` type. The type element specifies the member identifiers. This element has the following subelement:

- `symbols` - defines the member identifiers. Initial values for the members are specified by the `id` - `value` element.

Member identifiers are defined in the `moe` class of a symbol table that corresponds to the scope. When `enum` tag names exist, they are defined in a symbol table that corresponds to the scope.

Example

In "`enum  e1, e2, e3 = 10  ee;` " the `enumType` element corresponding to "`ee`" is as follows.

XcodeML

```
<enumType name="E0">
  <symbols>
    <id>
      <name>e1</name>
    </id>
    <id>
      <name>e2</name>
    </id>
    <id>
      <name>e3</name>
      <value><intConstant>10</intConstant></value>
    </id>
  </symbols>
</enumType>
```

3.8 Optional attributes for data type definition elements

The following are attributes for data type definition elements (these attributes can be omitted).

- `is_const` - specifies whether or not the variable defined for the data type is constant using `0` or `1` (`false` or `true`);
- `is_volatile` - specifies whether or not the variable defined for the data type is volatile using `0` or `1` (`false` or `true`);
- `is_restrict` - specifies whether or not the variable defined for the data type is restricted using `0` or `1` (`false` or `true`);

4 Symbol list

4.1 `id` element

The `id` element defines the variable names, array names, function names, `struct/union` member names, function arguments, and compound statement local variable names. The `id` element has the following attributes:

- `sclass` - represents one of the storage classes: '`auto`', '`param`', '`extern`', '`extern_def`', '`static`', '`register`', '`label`', '`tagname`', '`moe`', or '`typedef_name`';
- `type` - represents the identifier data type;

- `bit_field` attribute - specifies the member bit field in the `structType` and `unionType` elements;
- `is_gccThread` - specifies whether or not the GCC `_thread` keyword is defined using `0` or `1` (`false` or `true`); and
- `is_gccExtension` attribute.

The element has the following elements:

- `name` element - specifies the names of identifiers,
- `value` element - specifies the value corresponding to the identifier, and
- `bitField` element - specifies the member bit field in the `structType` and `unionType` elements.

If the identifier is a variable, it has an element for the address. However, there is no need for the address element if the variable is created by the compiler.

Example

The symbol table entry for the variable "`xyz`" in "`int xyz`" is as follows. It can be noted that `P6e7e0` is the `type_id` for "`int *`".

XcodeML
```
<id sclass="extern_def" type="int">
 <name>xyz</name>
 <value>
   <VarAddr type="P6e7e0">xyz</varAddr>
 </value>
</id>
```

The symbol table entry for the "`foo`" function in "`int foo()`" is as follows. It can be noted that `F6f168` is the `type_id` corresponding to the "`foo`" data type, and `P6f1a8` is the `type_id` of the pointer to `F6f168`. Furthermore, the `foo` identifier becomes the pointer to the function.

XcodeML
```
<id sclass="extern_def" type="0x6f168">
 <name>foo</name>
 <value>
  <funcAddr type="0xfla8">foo</funcAddr>
 </value>
</id>
```

4.2 `globalSymbols` element

Defines identifiers that have global scope. The element has `id` elements for identifiers with global scope.

4.3 `symbols` element

The symbols element defines identifiers that have local scope. The element has `id` elements that correspond to definition identifiers.

5 `globalDeclarations` element

The `globalDeclarations` element is used for declaring global variables in the program and defining functions. The element has the following elements:

- `functionDefinition` element - used to define functions,
- `varDecl` element - used to define variables,
- `functionDecl` element - used to declare functions, and
- `text` element - used to specify arbitrary text, such as directives.

5.1 `functionDefinition` element

The `functionDefinition` element is used to define functions. The element has the following elements:

- `name` element - specifies the function name,
- `symbols` element - specifies the parameter symbol list,

  The symbols element specifies the symbol table for the corresponding parameters.

- `params` element - used to define parameters, and
- `body` element - includes text for the `functionDefinition` element within the body of the function. The `functionDefinition` element within the body element indicates nested GCC functions.

  The element has the following attribute:

- `is_gccExtension` attribute.

5.2 `params` element

The `params` element specifies a list of parameters for the function.

- `name` element - specifies the name elements corresponding to the parameters.
- `ellipsis` - indicates variable length parameters. This element can be specified for the last subelement of the params element.
- The name elements within the `params` element must be ordered according to the parameter sequence. If there is data type information for a parameter, specify it using the `type` attribute of the `name` element.

5.3 `varDecl` element

The `varDecl` element is used to declare variables. Use the `name` element to specify the names of identifiers used in function declarations. The element has the following elements:

- `name` element - specifies the `name` element corresponding to the function to be declared.
- `value` element - specifies an initial value for the variable. Use the `value` element to specify initial values for arrays and structures, using multiple expressions.

Example

C code

```
int a[] = { 1, 2 };
```

XcodeML

```
<varDecl>
  <name>a</name>
  <value>
    <intConstant type="int">1</intConstant>
    <intConstant type="int">2</intConstant>
  </value>
</varDecl>
```

5.4 `functionDecl` element

The `functionDecl` element is used to declare functions. The element has the following element:

- `name` element - specifies the function name.

6 Statement element

This is an XML element that corresponds to the text syntax for the C language. The various elements have line number attributes added to them, which can be used to extract file information or the line number where the particular text is found.

6.1 `exprStatement` element

The `exprStatement` indicates a statement that is specified as an expression. The element has an expression element.

6.2 `compoundStatement` element

The `compoundStatement` element represents a compound statement. The element has the following elements:

- `symbols` element - a symbol list defined within the compound statement.
- `declarations` element - `varDecl`, `functionDefinition` or `functionDecl` elements that are associated with declarations found within the compound statement.
- `body` element - includes the main part of the compound statement.

6.3 `ifStatement` element

Element used for `if` statements. The element has the following elements:

- `condition` element - includes conditional expressions as elements,
- `then` element - includes the "`then`" portion as an element,
- `else` element - includes the "`else`" portion as an element.

6.4 whileStatement element

Element used for `while` statements. The element has the following elements:

- `condition` element - includes conditional expressions as elements and
- `body` element - includes the main statement portion as an element.

6.5 doStatement element

Element used for `do` statements. The element has the following elements:

- `body` element - includes the main statement portion as an element, and
- `condition` element - includes conditional expressions as elements.

6.6 forStatement element

Element used for `for` statements. The element has the following elements:

- `init` element - includes an initialization expression as an element,
- `condition` element - includes conditional expressions as elements,
- `iter` element - includes the iteration expression, and
- `body` element - includes the main body of the for statement.

6.7 breakStatement element

Element used for `break` statements. This is an empty element.

6.8 continueStatement element

Element used for `continue` statements. This is an empty element.

6.9 returnStatement element

Element used for `return` statements. The element has the `return` expression as an element.

6.10 gotoStatement element

Element used for `goto` statements. The element has either a `name` element or an expression as a subelement. The jump address for GCC can be specified in the expression with the following elements:

- `name` element - specifies the label name and
- `expression` - specifies the jump address value.

6.11 statementLabel element

Element used for the `goto` target label. The element has the label name as a `name` element.

- `name` element - specifies the label name

6.12 `switchStatement` element

Element used for `switch` statements. The element has the following elements:

- `value` element - specifies the `switch` value and
- `body` element - specifies the main body of the `switch` statement.

6.13 `caseLabel` element

Element used for the `case` statement in a `switch` statement. The element has the `case` value as an element.

- `value` element - specifies the `case` value

6.14 `gccRangedCaseLabel` element

Element used to specify the range in a GCC extension case statement. The element has the `case` value as an element.

- `value` element - specifies the lower limit of the `case` value.
- `value` element - specifies the upper limit of the `case` value.

6.15 `defaultLabel` element

Element used for the `default` label in a `switch` statement.

6.16 `pragma` element

The pragma element is used for the `#pragma` statement. The element has the character string that is used to specify the `#pragma` statement content.

6.17 `text` element

The `text` element contains arbitrary text. It is used to represent a string, such as a compiler-dependent directive, as an element. The element contains an arbitrary character string. This element also appears in `globalDeclarations`.

6.18 Line number attribute

All elements used for statements have attributes that indicate the line number and file name of the statement.

- `lineno` - has the value of the line number of the statement
- `file` - has the name of the file that contains the statement

7 Expression element

This is an XML element that corresponds to the expression syntax element in the C language. Each element has a data type with a `type` attribute associated with it that allows the data type information of the expression to be obtained.

7.1 Constant element

Constants can be expressed using the following elements:

- `intConstant` element - specifies a constant that has an integer value; The element describes a decimal or hexadecimal (starting from `0`) number.
- `longlongConstant` element - specifies two 32-bit hexadecimal (starting at `0`) numbers;
- `floatConstant` element - specifies a constant that has a `float`, `double` or `long double` value; the element describes a floating-point literal.
- `stringConstant` element - specifies a character string; the element has the attribute `is_wide="[1|0|true|false]"` (`0` if omitted). When `is_wide` is `1` or `true`, the character string has the `wchar_t` data type.
- `moeConstant` element - specifies an `enum` type constant; the element describes an `enum` constant.
- `funcAddr` element - specifies the address to a function. The element describes the function name.

The constant data type is specified using the `type` attribute. For `moeConstant`, the `moe` constant to be specified must be included in a symbol table that is within the scope of the expression.

7.2 Elements that reference variables

There is a different element for referencing each of the different types of variables: global variables, parameter variables, and local variables.

- `var` element - expression to reference global variables. The element specifies a variable name.
- `varAddr` element - expression to reference the address of a global variable. The element specifies a variable name.
- The scope attribute is used to differentiate local variables.
- `scope` attribute - has the value "`local`", "`global`" or "`param`".

XcodeML
```
<Var>var_name</Var>
```

is equivalent to

XcodeML
```
<PointerRef> <varAddr>var_name</varAddr></PointerRef>
```

XcodeML
```
<varAddr>var_name</varAddr>
```

and is written as `&var_name` in the C language.

7.3 `pointerRef` element

The `pointerRef` element is used to reference memory as an address expression for the subelement.

7.4 Elements for referencing array elements

The following elements are used for referencing arrays:

- `arrayRef` - expression that references the address of the first element in the array. The element specifies an array name.
- `arrayAddr` - expression that references the address of the array. The element specifies the array name.

To reference the array element, use `arrayRef` to calculate the address, and use `pointerRef` to access the element. In the same way as for variable references, the scope attribute is used to differentiate local variables.

7.5 Element for referencing structure members

When referencing structure members, use `memberAddr` for referencing the reference address, `memberRef` for referencing the member, and `memberArrayAddr` for referencing the member array address.

- `memberAddr` - references the address of a structure member; for the content of the element, specify the expression for the structure member address, and specify the member name for the member attribute value.
- `memberRef` - references a structure member; for the content of the element, specify the expression for the structure member address, and specify the member name for the member attribute value.
- `memberArrayAddr` - references the address of an array structure member; for the content of the element, specify the expression for the structure member address, and specify the member name for the member attribute value.
- `memberArrayRef` - references an array structure member; for the content of the element, specify the expression for the structure member address, and specify the member name for the member attribute value.

XcodeML
```
<memberRef memer="xxx">addr</memberRef>
```

is equivalent to

XcodeML
```
<pointerRef >
  <memberAddr memer="xxx">addr<memberAddr>
</pointerRef>
```

7.6 `assignExpr` element

The `assignExpr` element has two expressions as subelements and represents an assignment. The left expression (the first element) for the `assignExpr` element must be an lvalue.

7.7 Binary operation elements

The following elements represent binary arithmetic operations. The operands are specified as the content of two elements. The left expression is the first element, and the right expression is the second element.

- `plusExpr` - addition,
- `minusExpr` - subtraction,
- `mulExpr` - multiplication,
- `divExpr` - division,
- `modExpr` - remainder,
- `LshiftExpr` - shift left,
- `RshiftExpr` - shift right,
- `bitAndExpr` - bit-wise logical product (AND),
- `bitOrExpr` - bit-wise logical sum (OR), and
- `bitXorExpr` - bit-wise exclusive OR (XOR).

Taking the above and combining them with the assignment expressions yields the following elements:

- `asgPlusExpr` - addition,
- `asgMinusExpr` - subtraction,
- `asgMulExpr` - multiplication,
- `asgDivExpr` - division,
- `asgModExpr` - remainder,
- `asgLshiftExpr` - shift left,
- `asgRshiftExpr` - shift right,
- `asgBitAndExpr` - bit-wise logical product (AND),
- `asgBitOrExpr` - bit-wise logical sum (OR), and
- `asgBitXorExpr` - bit-wise exclusive OR (XOR).

For the above assignment operations, the left expression (the first element) must be an lvalue. The following elements represent logical binary arithmetic operations. The operands are specified as the content of two elements.

- `logEQExpr` - equivalence,
- `logNEQExpr` - nonequivalence,
- `logGEExpr` - greater than or equal to,

- `logGTExpr` - greater than,
- `logLEExpr` - less than or equal to,
- `logLTExpr` - less than,
- `logAndExpr` - logical product (AND), and
- `logOrExpr` - logical sum (OR).

7.8 Unary operation elements

The following elements represent unary arithmetic operations. The operand is specified as the content of an element.

- `unaryMinusExpr` - negation
- `bitNotExpr` - bit-wise negation (inversion)

The following element represents a logical unary arithmetic operation. The operand is specified as the content of an element.

- `logNotExpr` - logical negation (NOT)

The following elements represent the `sizeof` operator and the GCC extension operator:

- `sizeOfExpr` - `sizeof` operator; specifies an expression or the `typeName` element as a subelement.
- `gccAlignOfExpr` - represents the GCC `_alignof_` operator; specifies an expression or the `typeName` element as a subelement.
- `gccLabelAddr` - represents the GCC `&&` unary operator; specifies the label name.

7.9 `functionCall` element

The `functionCall` element represents a function call. The element has the following two elements:

- `function` element - specifies the address for the function that is called, and
- `arguments` element - specifies the arguments for the expression.

7.10 `commaExpr` element

The `commaExpr` element represents the comma expression (evaluated in sequence and returning the expression for the last element).

7.11 `postIncrExpr`, `postDecrExpr`, `preIncrExpr`, and `preDecrExpr` elements

The `postIncrExpr` and `postDecrExpr` elements represent the postincrement and postdecrement expressions in the C language. The content must be an lvalue. The `preIncrExpr` and `preDecrExpr` elements represent the pre-increment and predecrement expressions in the C language. The content must be an lvalue.

7.12 castExpr element

The castExpr element represents a type conversion (cast) expression or a compound literal.

The element has the following attributes:

- type attribute - specifies the type of the expression after conversion and
- is_gccExtension attribute.

The element has the following subelement:

- value - represents the literal portion of the compound literal.

7.13 condExpr element

The condExpr element corresponds to the ternary operator.

The element has three expressions as subelements, as shown in the example below.

XcodeML

```
<condExpr>
  (expression)
  (expression)
  (expression)
</condExpr>
```

7.14 gccCompoundExpr element

This element corresponds to the GCC extension compound expression. The element has the compoundStatement element.

- compoundStatement element - specifies the content of the compound expression

8 XcalableMP element

8.1 coArrayType element

Represents a co-array type declared by "#pragma xmp coarray". This element has the following attributes:

- type - derived data type name,
- element_type - co-array element data type name, and represents a co-array type of two or more dimensions when the type corresponding to the data type name is coArrayType.
- array_size - represents the co-array dimension.

This element has the following subelement:

- arraySize - represents the co-array dimension. When there is an arraySize element, the array_size attribute value should be "*".

Example

C code
```
int A[10];
#pragma xmp coarray [*][2]::A
```

The element that represents the type for variable `A` above is `coArrayType C2` given below.

XcodeML
```
<arrayType type="A1" element_type="int" array_size="10"/>
<coArrayType type="C1" element_type="A1"/>
<coArrayType type="C2" element_type="C1" array_size="2"/>
```

8.2 `coArrayRef` element

The element represents a reference to a co-array type variable. This element has the following subelements:

- First expression - represents the co-array variable expression and
- Second expression - represents the expression for the co-array dimension. Specify more than one expression if there are multiple dimensions.

8.3 `subArrayRef` element

Represents a reference to a subarray. This element has the following subelements. Subelements cannot be omitted.

- The first element has the expression that represents the array.
- `lowerBound` - represents the lower limit of the index. This subelement has an expression element.
- `upperBound` - represents the upper limit of the index. This subelement has an expression element.
- `step` - represents the step size for the index. This subelement has an expression element.

9 Other elements and attributes

9.1 `typeName` element

The `typeName` element represents the name of the type. Various subelements are specified, such as `sizeOfExpr`, `gccAlignOfExpr`, and `builtin_op`. The attribute contains the type that indicates the data type name identifier.

9.2 `is_gccExtension` attribute

The `is_gccExtension` attribute defines whether or not the GCC `__extension__` keyword is added to the beginning of the element. The attribute has a value of `0` or `1` (`false` or `true`). The `is_gccExtension` attribute can be omitted, in which case it is the same as assigning a 0 value. The following elements can have the `is_gccExtension` attribute:

- `id`

- functionDefinition
- castExpr
- gccAsmDefinition

Example

The definition for "__extension__ typedef long long int64_t" corresponds to the following:

XcodeML
```
<id type="long_long" sclass="typedef" is_gccExtension="1">
  <name>int64_t</name>
</id>
```

9.3 gccAsm, gccAsmDefinition, and gccAsmStatement elements

The gccAsm, gccAsmDefinition, and gccAsmStatement elements define the GCC asm and __asm__ keywords. The elements have the asm variable string as an element.

- gccAsm - represents the asm expression. This element has the subelement given below:
  - stringConstant (1 item) - represents assembly code,
- gccAsmDefinition - represents the asm definition, The subelements are the same as for gccAsm.
- gccAsmStatement - represents the asm statement. The element has the following attribute:
  - is_volatile - indicates whether or not volatile is specified; uses 0 or 1 for false or true, respectively.

  The element has the following subelements:
  - stringConstant (1 item) - represents assembly code.
  - gccAsmOperands (2 items) - The first item represents the output operand, while the second represents the input operand. If the operands are omitted, a tag that does not have a subelement is described. The subelement has gccAsmOperand (multiple items) as a subelement.
  - gccAsmClobbers (0 -1 item) - represents clobber values. The subelement has zero or more stringConstant elements as subelements.
  - gccAsmOperand - represents the input and output operands. The element has the following attributes:
    * match (can be omitted) - represents the identifier specified instead of the matching constraint (corresponds to "[identifier]") and
    * constraint (can be omitted) - represents the constraint/constraint modifier.

    The element has the following subelement:
    * Expression (1 item) - represents the expression that defines whether the element is input or output.

Example

C code

```
asm volatile (
     "661:\n"
     "\tmovl %0, %1\n662:\n"
     ".section .altinstructions,\"a\"\n"
     ".byte %c[feat]\n"
     ".previous\n"
     ".section .altinstr_replacement,\"ax\"\n"
     "663:\n"
     "\txchgl %0, %1\n"
     : "=r" (v), "=m" (*addr)
     : [feat] "i" (115), "0" (v), "m" (*addr));
```

XcodeML

```
<gccAsmStatement is_volatile="1">
  <stringConstant><![CDATA[661:\n\tmovl ...]]></stringConstant>
  <gccAsmOperands>
    <gccAsmOperand constraint="=r">
      <Var>v</Var>
    </gccAsmOperand>
    <gccAsmOperand constraint="=m">
      <pointerRef><Var>addr</Var></pointerRef>
    </gccAsmOperand>
  </gccAsmOperands>
  <gccAsmOperands>
    <gccAsmOperand match="feat" constraint="i">
      <intConstant>115</intConstant>
    </gccAsmOperand>
    <gccAsmOperand constraint="m">
      <pointerRef><Var>addr</Var></pointerRef>
    </gccAsmOperand>
  </gccAsmOperands>
</gccAsmStatement>
```

9.4 gccAttributes element

The gccAttributes element defines the GCC __attribute__ keyword. The element has the __attribute__ argument character string. The gccAttributes element has multiple gccAttribute elements as subelements.

- Elements that represent a type all have the gccAttributes element as a subelement (0 - 1 item).
- The id element has the gccAttributes element as a subelement (0 - 1 item).
- The functionDefinition element has the gccAttributes element as a subelement (0 - 1 item).

Example

This example sets the gccAttributes subelements for an element that represents a type.

C code

```
typedef __attribute__((aligned(8))) int ia8_t;
ia8_t __attribute__((aligned(16)) n;
```

XcodeML

```
<typeTable>
   <basicType type="B0" name="int" align="8" size="4"/>
     <gccAttributes>
       <attribute>aligned(8)</attribute>
    </gccAttributes>
   </basicType>
   <basicType type="B1" name="int" align="16" size="4"/>
    <gccAttributes>
       <attribute>aligned(8)</attribute>
       <attribute>aligned(16)</attribute>
     </gccAttributes>
  </basicType>
 </typeTable>
<globalSymbols>
  <id type="B0" sclass="typedef_name">
     <name>ia8_t</name>
   </id>
  <id type="B1">
     <name>n</name>
   </id>
 </globalSymbols>
<globalDeclarations>
   <varDecl>
     <name>n</name>
   </varDecl>
 </globalDeclarations>
```

This example sets the gccAttributes of the subelements (0 - 1 item) for the id element and the functionDefinition element.

C code

```
void func(void);
void func2(void) __attribute__(alias("func"));

 void __attribute__((noreturn)) func() {
    ...
 }
```

XcodeML

```
<typeTable>
  <functionType type="F0">
```

```
    <params>
      <name type="void"/>
    </params>
  </functionType>
  <functionType type="F1">
    <params>
      <name type="void"/>
    </params>
  </functionType>
</typeTable>
<globalSymbols>
  <id type="F0" sclass="extern_def">
    <name>func</name>
  </id>
  <id type="F1" sclass="extern_def">
    <name>func2</name>
    <gccgccAttributes>
      <gccAttribute>alias("func")</gccAttribute>
    </gccgccAttributes>
  </id>
</globalSymbols>
<globalDeclarations>
  <functionDefinition>
    <name>func</name>
    <gccgccAttributes>
      <gccAttribute>noreturn</gccAttribute>
    </gccgccAttributes>
    <body>...</body>
  </functionDefinition>
</globalDeclarations>
```

9.5 builtin_op element

The builtin_op element represents a call to an intrinsic compiler. The element has the following elements, each having values from 0 to multiple items. The order of the subelements must match the order of the function arguments.

- expression - specifies the expression as an argument for the function that is called;
- typeName - specifies the type name as an argument for the function that is called; and
- gccMemberDesignator - specifies the structure or union member designator as an argument for the function that is called. The element has two attributes: ref, which indicates the structure or union derived data type name; and member, which indicates the member designator character string. The element has an expression for the array index (0 -1 item) and the gccMemberDesignator element (0 -1 item) as subelements.

9.6 is_gccSyntax attribute

The is_gccSyntax attribute defines whether or not the expression, statement, or declaration corresponding to a tag uses the GCC extension. The value is 0 or 1 (false or true). This attribute can be omitted, in which case it is the same as assigning a 0 value.

9.7 is_modified attribute

The is_modified attribute defines whether or not the expression, statement, or declaration corresponding to a tag is modified during compilation. The value is 0 or 1 (false or true). This attribute can be omitted, in which case it is the same as assigning a 0 value. The following elements can have the is_gccSyntax or is_modified attribute:

- varDecl,
- Statement elements, and
- Expression elements.

10 Code examples

Example 1

C code

```
int a[10];
int xyz;
struct {    int x;   int y;} S;
foo() {
        int *p;
        p =  &xyz;        /* Statement 1 */
        a[4] = S.y;        /* Statement 2 */
}
```

Statement 1:

XcodeML

```
<exprStatement>
  <assignExpr type=" P6fc98">
   <ponterRef type=" P6fc98">
      <varAddr scope="local" type="P70768">p</varAddr>
     </pointerRef>
    <varAddr type=" P70828">xyz</varAddr>
  </assignExpr>
</exprStatement>
```

or

XcodeML

```
<exprStatement>
  <assignExpr type=" P6fc98">
  <Var scope="local" type=" P6fc98">p</Var>
     <varAddr type=" P70828">xyz</varAddr>
 </assignExpr>
</exprStatement>
```

Statement 2:

XcodeML

```
<exprStatement>
 <assignExpr type="int">
    <pointerRef type="int">
      <plusExpr type=" P6fc98">
        <arrayAddr type=" P708e8">a</arrayAddr>
        <intConstant type="int">4</intConstant>
      </plusExpr>
   </pointerRef>
   <pointerRef type="int">
      <memberAddr type="P0dede" member="y">
         <varAddr type= "P70988">S</varAddr>
      </memberAddr>
  </pointerRef>
 </assignExpr>
</exprStatement>
```

or

XcodeML

```
<exprStatement>
 <assignExpr type="int">
    <pointerRef type="int">
      <plusExpr type=" P6fc98">
        <arrayAddr type=" P708e8">a</arrayAddr>
        <intConstant type="int">4</intConstant>
      </plusExpr>
   </pointerRef>
   <memberRef type="int" member="y">
      <varAddr type= "P70988">S</varAddr>
  </memberRef>
 </assignExpr>
</exprStatement>
```

XcodeML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XcodeProgram source="t3.c">
  <!--
    typedef struct complex {
        double real;
        double img;
    } complex_t;

    complex_t x;
    complex_t complex_add(complex_t x, double y);

    main()
    {
        complex_t z;
```

```
15
        x.real = 1.0;
        x.img = 2.0;

        z = complex_add(x,1.0);
20
        printf("z=(%f,%f)\n",z.real,z.img);

    }
    complex_t complex_add(complex_t x, double y)
25  {
        x.real += y;
        return x;
    }
  -->
30  <typeTable>
    <pointerType type="P0" ref="S0"/>
    <pointerType type="P1" ref="S0"/>
    <pointerType type="P2" ref="S0"/>
    <pointerType type="P3" ref="S0"/>
35  <pointerType type="P4" ref="S0"/>
    <pointerType type="P5" ref="F0"/>
    <pointerType type="P6" is_restrict="1" ref="char"/>
    <pointerType type="P7" ref="F2"/>
    <structType type="S0">
40    <symbols>
        <id type="double">
          <name>real</name>
        </id>
        <id type="double">
45        <name>img</name>
        </id>
      </symbols>
    </structType>
    <functionType type="F0" return_type="S0">
50    <params>
        <name type="S0">x</name>
        <name type="double">y</name>
      </params>
    </functionType>
55  <functionType type="F1" return_type="int">
      <params/>
    </functionType>
    <functionType type="F2" return_type="int">
      <params/>
60  </functionType>
    <functionType type="F3" return_type="S0">
      <params>
        <name type="S0">x</name>
        <name type="double">y</name>
```

```
      </params>
    </functionType>
  </typeTable>
  <globalSymbols>
    <id type="F0" sclass="extern_def">
      <name>complex_add</name>
    </id>
    <id type="S0" sclass="extern_def">
      <name>x</name>
    </id>
    <id type="F1" sclass="extern_def">
      <name>main</name>
    </id>
    <id type="F2" sclass="extern_def">
      <name>printf</name>
    </id>
    <id type="S0" sclass="typedef_name">
      <name>complex_t</name>
    </id>
    <id type="S0" sclass="tagname">
      <name>complex</name>
    </id>
  </globalSymbols>
  <globalDeclarations>
    <varDecl>
      <name>x</name>
    </varDecl>
    <funcDecl>
      <name>complex_add</name>
    </funcDecl>
    <functionDefinition>
      <name>main</name>
      <symbols>
        <id type="S0" sclass="auto">
          <name>z</name>
        </id>
      </symbols>
      <params/>
      <body>
        <compoundStatement>
          <symbols>
            <id type="S0" sclass="auto">
              <name>z</name>
            </id>
          </symbols>
          <declarations>
            <varDecl>
              <name>z</name>
            </varDecl>
          </declarations>
```

```
      <body>
        <exprStatement>
          <assignExpr type="double">
            <memberRef type="double" member="real">
              <varAddr type="P0" scope="local">x</varAddr>
            </memberRef>
            <floatConstant type="double">1.0</floatConstant>
          </assignExpr>
        </exprStatement>
        <exprStatement>
          <assignExpr type="double">
            <memberRef type="double" member="img">
              <varAddr type="P1" scope="local">x</varAddr>
            </memberRef>
            <floatConstant type="double">2.0</floatConstant>
          </assignExpr>
        </exprStatement>
        <exprStatement>
          <assignExpr type="S0">
            <Var type="S0" scope="local">z</Var>
            <functionCall type="S0">
              <function>
                <funcAddr type="P5">complex_add</funcAddr>
              </function>
              <arguments>
                <Var type="S0" scope="local">x</Var>
                <floatConstant type="double">1.0</floatConstant>
              </arguments>
            </functionCall>
          </assignExpr>
        </exprStatement>
        <exprStatement>
          <functionCall type="int">
            <function>
              <funcAddr type="F2">printf</funcAddr>
            </function>
            <arguments>
              <stringConstant>z=(%f,%f)\n</stringConstant>
              <memberRef type="double" member="real">
                <varAddr type="P2" scope="local">z</varAddr>
              </memberRef>
              <memberRef type="double" member="img">
                <varAddr type="P3" scope="local">z</varAddr>
              </memberRef>
            </arguments>
          </functionCall>
        </exprStatement>
      </body>
    </compoundStatement>
  </body>
```

```
    </functionDefinition>
    <functionDefinition>
      <name>complex_add</name>
      <symbols>
        <id type="S0" sclass="param">
          <name>x</name>
        </id>
        <id type="double" sclass="param">
          <name>y</name>
        </id>
      </symbols>
      <params>
        <name type="S0">x</name>
        <name type="double">y</name>
      </params>
      <body>
        <compoundStatement>
          <symbols>
            <id type="S0" sclass="param">
              <name>x</name>
            </id>
            <id type="double" sclass="param">
              <name>y</name>
            </id>
          </symbols>
          <declarations/>
          <body>
            <exprStatement>
              <asgPlusExpr type="double">
                <memberRef type="double" member="real">
                  <varAddr type="P4" scope="param">x</varAddr>
                </memberRef>
                <Var type="double" scope="param">y</Var>
              </asgPlusExpr>
            </exprStatement>
            <returnStatement>
              <Var type="S0" scope="param">x</Var>
            </returnStatement>
          </body>
        </compoundStatement>
      </body>
    </functionDefinition>
  </globalDeclarations>
</XcodeProgram>
```