# *Chapter 6.*
# Types of Reference Information

Topic: Ignore

This chapter describes how reference information is presented in the ACIS documentation set. Most reference information is for *code items*, however, other types of reference information may also be presented.

Types of code item reference information include:

- Class
- Coclass (Class)
- Enumeration
- Function
- Interface (Class)
- Law Symbol
- Option
- Scheme Data Type
- Scheme Extension
- Shader
- Typedef

Other types of reference information include:

- Component

Each type of reference information is presented in a specific format called a *template*. Each template contains specific named data fields of information which apply to that reference type. This chapter contains *explanatory templates* that describe the format and contents of the different types of templates. The explanatory templates are listed in alphabetical order by type name.

## Class Template

Type: Ignore

**<NAME>** . . . . . The *Name* title indicates the exact string used to identify the C++ class (names are case sensitive).

Purpose: The *Purpose* field summarizes the intended purpose of the class.

Derivation: The *Derivation* field specifies the derivation of the class. The class described by the template is always listed on the left, followed by its parent, grandparent, etc. The last class in the list is always a base class, indicated by a trailing hyphen (–). Classes with no parents (base classes) show only the class name and a trailing hyphen (–).

SAT Identifier: The *SAT Identifier* field contains either a quoted identifier string or the word "None".

- The *quoted identifier string* is the word relating to the class that is written to the SAT save file in addition to the saved data for this class. This string is the same as that returned by the class method type_name.
- The word "None" in this field means there is no identifier written to the save file for this class. However, the class may have data that is written to the SAT save file.

Filename: The *Filename* field specifies the name of the header file, including the directory path, containing the prototype of the class. To use this class, applications should include this header file. The first element of the file path is the top-level installation directory for the component.

Description: The *Description* field describes the class, its use, and its structure. This field may provide other general information that applies to the entire class.

Limitations: The *Limitations* field describes any limitations of the class. These can include conceptual limitations, such as operations for which the class should not be used, real limitations, such as operations the class does not perform, or platform-specific limitations.

***Note*** *The use of term limitations here does not equate to a legal performance standard, rather the term limitation means "additional guidance" regarding the usage.*

References: The *References* field lists classes that this class references and classes by which this class is referenced. If the data of the class described in this template contains a pointer (often private) to another class, this template's class "references" that other class. If another class contains a pointer to this template's class, the class described in this template is "referenced by" the other class.

Component names, shown at the left, show the component in which the referenced classes are defined. If the component name is preceded by the word "by," then this template's class is *referenced by* that class(es).

Data: The *Data* field describes public and protected data items used by the class. Public data can generally be accessed by any other function. Protected data can generally be referenced only by this class and derived classes. Each data item is shown explicitly scoped, although in the actual code it may be block-scoped or may be defined by a preprocessor macro. Private data is never shown.

Constructor: The *Constructor* field lists public and protected constructors. The default C++ constructor generally creates an instance by reserving space without initializing the data. Other constructors may initialize the instance using arguments or may copy the data from other instances.

Destructor: The *Destructor* field lists public and protected destructors. If a lose destructor is provided, it should be used instead of any other destructors because it handles instances that can be used by multiple users.

Methods: The *Methods* field lists, in alphabetical order, the public and protected class methods (member functions). Certain methods common to all classes derived from ENTITY are not shown here; these methods are defined uniquely for each class using the ENTITY_FUNCTIONS macro. Because these methods are rarely called by applications, they are documented only in the ENTITY class template.

Internal Use: The *Internal Use* field alphabetically lists class methods that are internal to ACIS and not intended for direct usage. These are public or protected class methods which are declared in the class header file and may be needed for class derivation. If required, refer to the header file and other sources for more detailed information.

Related Fncs: The *Related Fncs* field lists other related functions declared in the class header file that are not class methods.

# Coclass Template

Type: Ignore

**<NAME>** . . . . . The *Name* title is a case sensitive string used to identify the C++ coclass. Coclasses are used in conjunction with C++ interface classes, called simply interfaces.

Interfaces are used for coclass communication. Interfaces are the services. Coclasses don’t care about other coclasses, but rather they care about the services. Interfaces are pure abstract base classes. They provide the definition of the interface only.

Purpose: The *Purpose* field summarizes the intended purpose of the coclass.

Derivation: The *Derivation* field specifies the derivation of the coclass. The class described by the template is always listed on the left, followed by its parent, grandparent, etc. The last class in the list is always a base class, indicated by a trailing hyphen (–). Classes with no parents (base classes) show only the class name and a trailing hyphen (–).

GUID: The *GUID* field specifies the Globally Unique Identifier for the coclass. This is a long hexidecimal number that uniquely identifies the interface to the operating system. It is used in Microsoft COM to provide the handle to this coclass.

SAT Identifier: The *SAT Identifier* field contains either a quoted identifier string or the word "None".

- The *quoted identifier string* is the word relating to the class that is written to the SAT save file in addition to the saved data for this class. This string is the same as that returned by the class method type_name.
- The word "None" in this field means that there is no identifier written to the save file for this class. However, the class may have data that is written to the SAT save file.

Filename: The *Filename* field specifies the name of the header file, including the directory path, containing the prototype of the class. To use this class, applications should include this header file. The first element of the file path is the component's top-level installation directory.

Description: The *Description* field describes the class, its use, and its structure. This field may provide other general information that applies to the entire class.

Data: The *Data* field describes public and protected data items used by the coclass. Public data can generally be accessed by any other function. Protected data can generally be referenced only by this class and derived classes. Private data is never shown.

Interfaces: The *Interfaces* field lists the interface classes that this coclass references. The interface classes define the services that are available to a coclass object and specify the manner in which those services are accessed.

Constructor: The *Constructor* field lists public and protected constructors. Generally, coclass constructors are never called directly. Instead, class factories are used to create the coclass object instance.

Destructor: The *Destructor* field lists public and protected destructors. Generally, coclass destructors are never called directly. Instead release tools are provided which clean up associated data and pointers.

Methods: The *Methods* field alphabetically lists the public and protected coclass methods (member functions) which are not already specified by an interface class and are not already marked for internal use only.

Internal Use: The *Internal Use* field alphabetically lists class methods that are internal to ACIS and not intended for direct usage. These are public or protected coclass methods which are declared in the class header file, are not already specified by an interface class, and may be needed for coclass derivation. If required, refer to the header file and other sources for more detailed information.

Related Fncs: The *Related Fncs* field lists other useful functions declared in the class header file that are not class methods. If the class has friend operator related functions, their prototypes are provided after this list.

# Component Template

Type: Ignore

**<NAME>** . . . . . The *Name* title indicates the name of the component. A *component* is a specialized collection of software items (classes, functions, etc.) that are grouped together to serve a distinct purpose, such as rendering or blending. Components are the means by which *Spatial* breaks down the ACIS products into manageable pieces by functionality.

The component “template” marks the beginning of every component manual and introduces the component. This template does not contain named data fields (such as a *Description* field) like other templates. However, this introduction to each component does contain a standard set of information, including the full and abbreviated component names, a summary of the intended purpose of the component, and the associated top-level directory name for the component. This top–level component directory generally contains several subdirectories. It may contain:

– One or more subdirectories that contain the source files that implement functions, classes, etc. for the component
– A subdirectory containing files that support the Scheme interface, if applicable (*_scm)
– A subdirectory containing object files (obj)

# Enumeration Template

Type: Ignore

**<NAME>** . . . . . The *Name* title indicates the exact string used to identify the enumeration. In C++, an enumeration (enumerated data type) defines a list of constant integer values, each associated with a unique identifier name. Enumerations are defined using the enum keyword. An enumeration template may document an item that is defined in one of the following ways:

- Using the enum keyword
- Using the typedef enum keyword combination
- Using the enum_entry structure

The ACIS enum_entry is used to define a structure that is technically not a C++ enumerated data type, but is similar. It maps a character string to an integer value (this value is often specified using the identifier from a previously defined enum). The enum_entry structure is internal to ACIS.

Purpose: The *Purpose* field summarizes the intended purpose of the enumeration.

Filename: The *Filename* field specifies the name of the source file, including the directory path, containing the definition of the enumeration. The first element of the file path is the component's top-level installation directory.

Value: The *Value* field lists the possible values for the enumeration. The left column contains the identifier and the right column contains the description. If this was defined using the enum_entry structure, the right column contains the string associated with each identifier.

# Function Template

Type: Ignore

**<NAME>** . . . . . The *Name* title indicates to the reader the exact string used to identify the function. This template documents API and DI functions.

Action: The *Action* field summarizes what the function does.

Prototype: The *Prototype* field shows the function's prototype. If the function is overloaded, all the prototypes are provided.

Includes: The *Includes* field lists the #include files that are needed to define the function's return type, the function's prototype(s), and the data types used in the argument list.

The header files for return types and data types are only those needed to define the ACIS classes, enumerations, and typedefs used in the function prototype. If a data type is defined in some other way, such as using a #define statement, the *Includes* field may not contain the header file that defines that data type. For example, some functions contain references to the data type logical, which is defined as equivalent to an int in the file logical.h using a #define statement. The *Includes* field does not contain the file logical.h in such cases.

If the function is overloaded, the *Includes* list is cumulative for all the function's prototypes. The standard ACIS header file, acis.hxx, is included in all lists.

Description: The *Description* field describes the function, its use, and its surrounding conditions.

Errors: The *Errors* field lists any error conditions that may occur when the function is called.

Limitations: The *Limitations* field lists any limitations of the function. These can include conceptual limitations, such as operations for which the function should not be used, real limitations, such as conditions the function does not support, or platform-specific limitations.

***Note*** *The use of term limitations here does not equate to a legal performance standard, rather the term limitation means "additional guidance" regarding the usage.*

Library: The *Library* field lists the "base name" of the object library file in which the function is defined. The full object library filename is platform and installation dependent. Refer to the chapter *Object Libraries* in the *3D ACIS Application Development Manual* to determine the full name from the base name.

Filename: The *Filename* field specifies the name of the source file, including the directory path, containing the implementation of the function. The first element of the file path is the component's top-level installation directory.

Effect: The *Effect* field specifies how the function influences the model:

- "Changes model" means the function causes a change in the model.
- "Read-only" means the function is an inquiry function that does not change the model.
- "System routine" means the function performs a system-level function that does not change the model.

# Interface Template

Type: Ignore

**<NAME>** . . . . . The *Name* title is a case sensitive string used to identify the C++ interface class, called simply interfaces. Interfaces are used in conjunction with C++ coclasses, called simply objects.

Interfaces are used for object communication. Interfaces are the services. Objects don't care about other objects, but rather they care about the services. Interfaces are pure abstract base classes. They provide the definition of the interface only.

Purpose: The *Purpose* field summarizes the intended purpose of the interface class.

Derivation: The *Derivation* field specifies the derivation of the interface class. The class described by the template is always listed on the left, followed by its parent, grandparent, etc. The last class in the list is always a base class, indicated by a trailing hyphen (–). Classes with no parents (base classes) show only the class name and a trailing hyphen (–).

GUID: The *GUID* field specifies the Globally Unique Identifier for the interface class. This is a long hexidecimal number that uniquely identifies the interface to the operating system. This is used in Microsoft COM to provide the handle to this interface class.

Filename: The *Filename* field specifies the name of the header file, including the directory path, containing the prototype of the class. To use this class, applications should include this header file. The first element of the file path is the component's top-level installation directory.

Description: The *Description* field describes the class, its use, and its structure. This field may provide other general information that applies to the entire class.

Coclasses: The *Coclasses* field lists the coclasses that reference this interface class. The interface classes define the services that are available to coclass objects and specify the manner in which those services are accessed.

Methods: The *Methods* field alphabetically lists the interface class methods (member functions). The interface defines the services that are available to a coclass object, while the interface methods specify the manner in which those services are accessed.

# Law Symbol Template

Type: Ignore

**<NAME>** . . . . . The *Name* title indicates the name string used to identify the law symbol; however, this is not necessarily the string that is entered when using the law symbol.

The *Name*s are presented in uppercase letters. In the default law parsing mode, law symbol names are not case sensitive and are converted in the internal representation to uppercase.

Several law symbols contain only symbolic characters, such as +, –, /, *, and ^, and not alphanumeric characters. In these cases, the *Name* is in lowercase letters and is not the actual symbolic character(s), but rather a string that relates to the class used to implement the law symbol. The *Syntax* field shows the actual symbolic character(s) to use.

The *Name* constant, referring to the C++ constant_law class, refers to a constant integer or real number. The *Syntax* field shows the character #, which means a number (generally an integer) should be entered in its place.

The symbol # is also used in the *Name* field to indicate that an integer must be used as part of the law symbol. For example, A# (from the identity_law class) means that A0, A1, etc. is entered when using the law symbol. Similarly, EDGE#, WIRE#, and SURF# require integers as part of their usage.

Action: The *Action* field summarizes what the law symbol does.

Derivation: The *Derivation* field specifies the class and its derivation for the given law symbol. The class for the law symbol is always listed on the left, followed by its parent, grandparent, etc. The last class in the list is always a base class, indicated by a trailing hyphen (–). Classes with no parents (base classes) show only the class name and a trailing hyphen (–).

Syntax: The *Syntax* field describes how the law symbol is entered by the user. The following font and punctuation conventions are used to represent different components of the syntax:

**Bold** Bold-faced letters denote the law symbol name. Nonbold-faced letters denote arguments to the law symbol, if applicable. The arguments are other law symbols. The law symbol name is presented in uppercase. However, in the default law parsing mode, law symbol names are not case sensitive.

( ) Parentheses surround all of the arguments to the law symbol.

[ ] Square brackets represent optional syntax elements.

. . . Horizontal ellipsis points indicate the omission of several items in a list. In the following example, the law symbol takes a list of two or more laws and one or more step numbers as arguments:

**STEP** (<law1>, num1, ..., <law>, num)

# Indicates that a number should be entered in place of the #.

Description: The *Description* field describes the law symbol, its use, its arguments, the dimensions of its arguments, and the surrounding conditions.

Example: The *Example* field contains a Scheme example that illustrates how the law symbol can be used. The law symbol may be used in combination with others to build a law function string. When the string containing the law function is passed into the law Scheme extension (which defines a law Scheme data type), it is enclosed within double-quotation marks. The quotation marks surround only the outer-most law symbol in the law function definition.

Comment lines are indicated in the Scheme example by a leading single semicolon (;). Text lines returned as output by the extension are indicated by leading double semicolons (;;). Refer to the *Scheme Extension Template* description in the *3D ACIS Online Help User's Guide* for more information on Scheme examples.

# Option Template

Type: Ignore

**<NAME>** . . . . . The *Name* title identifies the option. Options may be set to modify the behavior of ACIS.

Action: The *Action* field summarizes what the option does.

Name String: The *Name String* field indicates the string used to refer to the option in Scheme extensions or APIs. The minimum required portion of the string is shown in bold. The string may be abbreviated down to this minimum required portion when used in Scheme extensions or APIs.

Options may be set in a Scheme application (such as Scheme AIDE) using the Scheme extension option:set; or in a C++ application using an API function.

Scheme: The *Scheme* field contains three columns for the type, valid values, and initial value of the option when used in a Scheme extension. The type is a Scheme data type, either boolean, integer, string, or real. Many options are simply flags indicating an *on* or *off* state. These options are booleans, with a value of #t indicating *on* and a value of #f indicating *off*. If the option does not apply to the Scheme environment, this field contains the words "Not applicable."

C++: The *C++* field contains three columns for the type, valid values, and initial value of the option when used in C++. The type is int, double, char*, or logical (which is defined in ACIS and is equivalent to an int). Many options are simply flags indicating an *on* or *off* state. These options are logicals, with a value of TRUE indicating *on* and a value of FALSE indicating *off*. If the option does not apply to the C++ environment, this field contains the words "Not applicable."

There are 3 APIs for setting options. The appropriate API to use depends on the type of the value: api_set_int_option when the value is an int (or, equivalently, a logical); api_set_dbl_option when the value is a double; and api_set_str_option when the value is a char* (character string).

Description: The *Description* field describes the option and its use.

Example: The *Example* field contains a Scheme example that sets the option (possibly including surrounding calls that provide context). Comment lines are indicated by a leading single semicolon (;). Text lines returned as output by the extension are indicated by leading double semicolons (;;). Refer to the *Scheme Extension Template* description in the *3D ACIS Online Help User's Guide* for more information on Scheme examples.

If the option does not apply to the Scheme environment, this field contains the words "Not applicable."

When appropriate, an illustration follows the example.

# Scheme Data Type Template

Type: Ignore

**<NAME>** . . . . . The *Name* title indicates the exact string used to identify the Scheme data type. ACIS provides Scheme data types (Scheme objects) specifically for use with ACIS Scheme extensions, in addition to those native to the Scheme language.

Description: The *Description* field describes the data type and its use and its surrounding conditions.

Derivation: The *Derivation* field specifies the derivation of the Scheme data type. The data type described by this template is always listed on the left, followed by its parent, grandparent, etc. All data types are ultimately derived from the base type scheme–object.

C++ Type: The *C++ Type* field indicates the exact string used to identify the C++ class associated with this Scheme data type.

External Rep: The *External Rep* field indicates the exact string that is displayed in returned information to represent this data type. Each variable of the external representation is described.

Example: The *Example* field shows a Scheme example that illustrates the use of the data type. Comments are indicated by a single semicolon (;). Text returned by the extension is indicated by a double semicolon (;;). Refer to the *Scheme Extension Template* description in the *3D ACIS Online Help User's Guide* for more information on Scheme examples.

# Scheme Extension Template

Type: Ignore

**<NAME>** . . . . . The *Name* title indicates the exact string used to identify the Scheme extension.

Action: The *Action* field summarizes what the Scheme extension does.

Filename: The *Filename* field specifies the name of the source file, including the directory path, containing the implementation of the Scheme extension. The first element of the file path is the component's top-level installation directory.

APIs: The *APIs* field indicates the API functions that are called by the extension. These are determined by running the example (refer to the *Example* field) through the Scheme AIDE demonstration application and extracting the names of the API functions called by the extension.

This is done by preceding the call to the extension of interest with the debug:module extension, specifying module name "api" and level "calls" as arguments. The resulting list of APIs called is redirected to an output file using the debug:file extension with a filename argument. API functions called by other extensions in the example are ignored, and are not recorded in this field. Other examples with different surrounding conditions or data may produce a different list of APIs.

Syntax: The *Syntax* field describes how the Scheme extension command is entered by the user. Each Scheme extension command is divided into one or more syntax tokens, which may be literals, such as command names or keywords, arguments, or compound tokens. The following font and punctuation conventions are used to represent different components of the syntax:

( ) Parentheses surround the entire Scheme extension command.

**Bold** Boldfaced letters denote a syntax literal such as a command name, an option, or a keyword. Keywords that combine bold and normal fonts can be abbreviated to the letters in bold or they may be entered completely. Nonbold arguments are variables that the user determines at the time the command is entered, and whose type is given in the *Arg Types* field.

`[]` Square brackets represent an optional block of syntax elements.

`|` A vertical bar (logical "or") separating syntax tokens indicates that one or the other token should be specified.

`&|` An ampersand and vertical bar pair (logical "and/or") separating syntax tokens indicates one or both tokens may be specified.

`{}` Braces represent a block of syntax elements.

`{}*` Braces and a superscript asterisk represent a block of syntax elements that is repeated zero or more times. In the following example, zero or more bulge and position pairs must be specified:

```
(wire-body:kwire [vector] position
   {bulge position}*)
```

`{}+` Braces and a superscript plus sign represent a block of syntax elements that is repeated one or more times.

`{}n` Braces and a superscript count modifier represent a block of syntax elements that is repeated exactly *n* times, where *n* may refer to syntax elements that are specified elsewhere in the command.

`...` Horizontal ellipsis points indicate the omission of several optional items in a list. In the following example, the Scheme extension command takes a list of two or more bodies as arguments:

```
(bool:unite body1 ... bodyn)
```

Arg Types: The *Arg Types* field lists each argument used in the *Syntax* field, followed by the type of the argument. Standard Scheme syntax containing parentheses is used for describing a "list" such as (entity . . .), or a "pair" of items such as (entity . position). Alternative types are separated by a | "or" symbol, such as (entity | (entity . . .), which means the argument may be a single or list of entities.

Returns: The *Returns* field indicates the type of Scheme object returned by the extension. Returned objects are absorbed by nesting Scheme calls, or their external representation is printed if they are not absorbed.

Errors: The *Errors* field lists any common errors that may occur when using this extension.

Description: The *Description* field describes the extension, its use, and the surrounding conditions.

Limitations: The *Limitations* field lists any limitations to the extension. These can include conceptual limitations, such as operations for which the extension should not be used, real limitations, such as conditions the extension does not support, or platform-specific limitations.

***Note*** *The use of term limitations here does not equate to a legal performance standard, rather the term limitation means "additional guidance" regarding the usage.*

Example: The *Example* field shows an example of the extension and any necessary surrounding Scheme statements that provide context. The example may be followed by an illustration.

Comment lines are indicated by a leading single semicolon (;). Text lines returned as output by the extension are indicated by leading double semicolons (;;).

In the case of define statements, the double semicolon output lines may include the result of the define followed by "=>" and its assumed entity number. The entity number is helpful to know, because subsequent Scheme extensions output the entity number and not the Scheme variable name. However, the actual entity number may be different in your Scheme session. This is why the Scheme examples use the defined variable name as input to subsequent statements. This makes the example more general and useful for copying and pasting into your Scheme session. The actual entity number from your Scheme session could alternatively be used as input instead of the variable name.

The double semicolon is also used in Scheme journal files to indicate returned objects or text from the extension.

Hexadecimal pointer values returned in external representations are shown as %x, rather than as the actual hex value, which can vary from run to run.

Scheme examples used in the documentation are created and tested by *Spatial*. All options are set to the default values, unless modified in the example. Figures that are reproduced in the documentation are created using the dl:interleaf or render:interleaf extensions. The figure is scaled as needed for reproduction in the manual.

Prior to running the examples, the Scheme file manual.scm (shown below) is loaded to provide an isometric view, set the display background, etc. (*Spatial* loads this file by adding a load command to the acisinit.scm initialization file. Refer to the *3D ACIS Getting Started Guide* for more information about initialization files.)

```
;------------------------------------------------
; manual.scm
;
; This file is used to set up Scheme AIDE for
; creating and testing examples for documentation.
; It creates a view, changes the default view from a
; front to an iso view, etc.
;
; To put an image in the Interleaf document: render
; the image to a file, open the file as a document,
; size the image to 0.29, and paste the image into a
; window frame.
;------------------------------------------------

(define my_view (view:dl 800 0 545 600))

(view:set (position 100 –200 100)
   (position 0 0 0)
   (gvector 0 0 1) my_view)

(view:refresh my_view)

; For Interleaf purposes change the render
; background to white.

(define bg1 (background ”plain”))
(define rdsb (render:set–background bg1))
(define bsp (background:set–prop bg1
   ”color” (color:rgb 1 1 1)))
```

```
; Set roll back state to return to between examples.

(roll:name-state "begin")

; Since all rendered images in the documentation are
; created (using Advanced Rendering) with the
; render:interleaf extension, the following is used.

(view:interleaf #t)
(render:set-mode "full")

;------------------------------------------------
```

# Shader Template

Type: Ignore

**<NAME>** . . . . . The *Name* title indicates the name for the specific type of shader.

Action: The *Action* field summarizes what the shader does.

Classification: The *Classification* field lists the classification (the top level of shader hierarchy) of the shader. Classifications include background, foreground, color, displacement, reflectance, transparency, texture space, or light.

Component: The *Component* field specifies the name of the software component that contains the source file for the shader implementation.

Arguments: The *Arguments* field lists each shader argument name, type, and default value for all arguments that are effective in this renderer. These arguments are used to control the rendered image produced by this renderer and can be inquired, set, saved to a SAT file, etc.

This renderer may use these arguments to produce results different from the results of other renderers. For example, one renderer may use the light color and dark color arguments of a shader to produce an image simulating wood grain, with light wood and dark wood patterns. Another renderer may simply use these two color arguments to produce one solid intermediate color for the image.

Additional Args: The *Additional Args* field lists each shader argument name, type, and default value for additional arguments that apply to this shader type. This argument list is used by the specific renderer only to ensure an interface common to all renderers. This means that these arguments *have no effect* on the rendered image produced by this renderer, but can be inquired, set, saved in a SAT file, etc. for use by other renderers.

Description: The *Description* field describes the shader, its usage, and its arguments.

Example: The *Example* field shows a Scheme example that illustrates the use of the shader. Comments are indicated by a single semicolon (;). Text returned by the extension is indicated by a double semicolon (;;). Refer to the *Scheme Extension Template* description in the *3D ACIS Online Help User's Guide* for more information on Scheme examples.

# Typedef Template

Type: Ignore

**<NAME>** . . . . . The *Name* title indicates the exact string used to identify the typedef. In C++, a typedef (type name definition) is used to define a new data type name. A typedef declaration does not create a new data type; it simply creates a new name (synonym) for an existing type. Typedefs are useful for making programs more portable and more readable.

Typedefs are declared using the typedef keyword. (A data type synonym that is defined with the typedef enum keyword combination is not documented as a typedef, but rather as an enumeration in an Enumeration Template.)

Purpose: The *Purpose* field summarizes the intended purpose of the typedef.

Filename: The *Filename* field specifies the name of the source file, including the directory path, containing the definition of the typedef. The first element of the file path is the component's top-level installation directory.

Definition: The *Definition* field provides the C++ statement that defines the typedef.

Description: The *Description* field describes the typedef.