

## **Pathfinder Solutions**

90 Oak Point Wrentham, Massachusetts 02093 U.S.A



web: [www.pathfindersol.com](http://www.pathfindersol.com)  
voice: +01 508-384-1392  
fax: +01 508-384-7906

# Using Realized Types in a Multi-Processor Design

5/1/04

Carolyn K. Duby  
version 1.2

copyright entire contents 1995 - 2004 **Pathfinder Solutions** all rights reserved

## Motivation

When integrating with realized domains it is often necessary to hold a handle to a realized type that must be sent as a parameter to a service or set as a parameter to a service handle. This paper describes ways to extend the Pathfinder design to incorporate realized types passed in service calls or service handles across process boundaries.

## Defining a Realized User-Defined Type

To define a realized type, open the type file for the appropriate scope. The type file for the system is located in <system\_encyclopedia>\types\<system\_name>.typ. The type file for a domain is located in <domain\_encyclopedia>\types\<domain\_prefix>.typ. Add a user-defined type based on the built-in Handle type as follows:

```
EXTERN TYPE <type_name> Handle {ReferencedType = "<base_type>"
Serializeable = "T" ReferenceCounted = "T"}
```

The keyword **EXTERN** indicates that springboard will recognize the type and treat it as a Handle. However, the templates will not generate a definition of the type. The definition must be declared in the sys\_def.hpp file or a file containing the definition must be included in the sys\_def.hpp file. The <type\_name> is the name of the type as it will be used in the analysis models.

Extern types that are sent across processor require a few properties to be defined in {} after the type definition. The **ReferencedType** property is the type that will be allocated when a new instance of the handle is created. For example, the templates will need to create a new instance of the handle when the realized type is sent across the process boundary. The templates will generate code similar to the following:

```
<type_name> new_handle = new <base_type>;
```

The **Serializeable** property equal to "T" indicates that the type may be serialized and passed across processor boundaries and to Instrumented Execution. Serializeable equal to "F" means that the type may not be passed across processor boundaries and will be shown in Instrumented Execution as an address. If the Serializeable property is "T", the type must map to a pointer to a class derived from either PfdRCSerializeable, for reference counted handles, or PfdSerializeable, otherwise. See the description of the ReferenceCounted property for more information.

The **ReferenceCounted** property indicates if the realized type is reference counted. Reference counting is a mechanism for deallocating the memory associated with a handle when there are no more references to the handle. Reference counting avoids memory leaks when using realized types. Alternative ways to deal with memory cleanup are calling an explicit realized service to deallocate the handle when it is no longer needed or defining the Explicit Delete property of an attribute. A value of "T" indicates that the type is reference counted and will be accessed through a smart pointer. For a discussion of smart pointers, consult Scott Meyers' book *More Effective C++*, Item28. If the ReferenceCounted property is "F", the type will be accessed through a regular C++ pointer. If ReferenceCounted is "T", the type must be a pointer derived from the class PfdRCSerializeable. If ReferenceCounted is "F", the type must be a pointer derived from PfdSerializeable.

## Implementing a Realized Type

- Declare a new class derived from PfdSerializeable, if not using reference counting, or PfdRCSerializeable, if reference counting.
- Add a default constructor to the derived class. The default constructor will be called when a new instance of the class is allocated. For example, when deserializing a realized instance when it is received as part of a service handle.

- Implement the pure virtual methods:

```
// serialize - save the realized type to a buffer
virtual void insertToBuffer (message_buffer_t *msg_buf, int *msg_len, bool_t is_ascii = FALSE);

// create a new instance of the realized type and call internalExtractFromBuffer to deserialize it
static RealizedType* extractFromBuffer(message_buffer_t* msg_buf,int *msg_len,
                                      bool_t is_ascii);
// deserialized - populate this instance with the values from the given buffer
virtual bool_t internalExtractFromBuffer(message_buffer_t* msg_buf,int *msg_len,
                                      bool_t is_ascii);
// virtual copy constructor - dynamically allocate a new instance of the realized type and
// call the copy constructor
virtual PfdSerializeable* duplicate();

#ifdef NO_PATH_IE
// convert this instance to its string representation as displayed in Spotlight – This function is
// called to show a realized type in the Spotlight watch window
virtual void convertToString(PfdString& str_value);
// set the values of this instance from its string representation in Spotlight – This function is called
// when the value of a realized type is changed in the Spotlight watch window. Return the position
// of the first character in the next token
virtual void readFromString(const PfdString& str_value, int& curr_char);
#endif
```

- Add a typedef to define the user defined type to a pointer to the realized type. For example  

```
typedef RealizedType* RealizedTypeHandle;
```

where RealizedType is the name of the class derived from PfdSerializeable or PfdRCSerializeable and RealizedTypeHandle is the name of the extern user defined type referenced by the models and defined in the type file.

- Add other member functions and member variables necessary to implement your realized type.
- Add a realized domain to your Domain Chart with services that create and manipulate the realized type.
- For serialized types, create a smart pointer:  
In the header file:

```
BEGIN_DECLARE_SMART_POINTER(SmartRealizedTypeHandle, RealizedType)
DECLARE_CONVERSION(PfdSmartSerializeable)
END_DECLARE_SMART_POINTER
```

In the .cpp file:

```
DEFINE_SMART_POINTER(SmartRealizedTypeHandle, RealizedType)
DEFINE_CONVERSION(SmartRealizedTypeHandle, PfdSmartSerializeable)
```