



VisiBroker® for Delphi



Inprise Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249

Refer to the file DEPLOY.TXT located in the root directory of your product for a complete list of files that you can distribute in accordance with the License Statement and Limited Warranty.

Inprise may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1983, 1999 Inprise Corporation. All rights reserved. All Inprise and Borland brand and product names are trademarks or registered trademarks of Inprise Corporation. Other brand and product names are trademarks or registered trademarks of their respective holders.

Printed in the U.S.A.

VisiBrokerForDelphi 1E0R899

9900010203-9 8 7 6 5 4 3 2 1

PDF

Contents

Chapter 1

Programmer tools 1-1

General information	1-1
idl2pas.	1-1
idl2ir.	1-3

Chapter 2

IDL to Pascal mapping 2-1

Names.	2-2
Reserved names	2-2
Reserved words	2-2
Modules.	2-3
Nested Modules	2-4
Basic types	2-5
IDL type extensions	2-6
Boolean	2-7
Char	2-7
Octet	2-7
String	2-7
WString	2-7
Integer types	2-7
Floating point types	2-8
Long Double types	2-8

Helper classes	2-8
Constants	2-9
Constants within an interface	2-9
Constants NOT within an interface	2-9
Constructed types.	2-10
Enum	2-10
Struct	2-10
Union	2-12
Sequence	2-13
Array	2-13
Interfaces.	2-14
Abstract Interfaces	2-14
Multiple Inheritance	2-15
Cross-Module Referencing	2-16
Passing parameters	2-18
Context Arguments to Operations	2-19
Mapping for exceptions	2-19
User-defined exceptions	2-19
Unknown user exceptions	2-20
System exceptions	2-20
Mapping for the Any type	2-22
Mapping for Typedef.	2-23

Index I-1



Tables

2.1	Basic type mappings	2-5
2.2	Summary of supported IDL extensions . .	2-6
2.3	IDL extensions for new types	2-6
2.4	IDL System Exception to Pascal Class Name Mapping	2-21

Programmer tools

This chapter describes the programmer tools offered by VisiBroker for Pascal. This chapter includes the following sections:

General information	page 1-1
idl2pas	page 1-1
idl2ir	page 1-3

General information

The VisiBroker programming tools are described in this chapter for a Windows environment.

To view options for a command, enter

Syntax command name -?

Example `idl2pas -?`

idl2pas

This command implements VisiBroker's IDL to Pascal compiler, which generates client stubs and server skeletons from an IDL file.

Syntax `idl2pas [arguments] infiles`

`idl2pas` takes one or more IDL files as input and generates the corresponding Pascal classes, including client stubs and server skeletons.

The `infiles` parameter lists the IDL file or files for which you want to generate Pascal code. The arguments provide control over the resulting code.

Example

```
idl2pas -hdr_suffix hx -server_ext _serv -no_tie -no_excep-spec bank.idl
```

The following table lists the arguments for `idl2pas`:

Argument	Description
-C -retain_comments	Retain comments that appear in the IDL file so that they appear in the generated Pascal code. The default is off. (pre-processor option)
-D<macro> -define <macro>	Define a macro (e.g. -DDebug), optionally with a value (e.g. -DVersion=2) (pre-processor option)
-H -list_includes	Print the full path names of included files to standard error (pre-processor option)
-h -help -usage -?	Print a description of all arguments to standard output.
-I<directory> -include<directory>	Add a directory to the include path. <directory> can specify a full or relative path. (pre-processor option)
-P -no_line_directives	Suppress generation of line number information in generated code. The default is off. (pre-processor option)
-U<macro> -undefine<macro>	Undefine a specified macro. (pre-processor option)
-comments	Add comments to generated files. The default is on. Do not use with -no_comments.
-generate_empty_units	Create unit files even if they are empty. The default is off. Do not use with -no_generate_empty_units.
-generate_implementation_units	Create Pascal server-side implementation units. The default is on. Do not use with -no_generate_implementation_units.
-generate_interface_units	Create Pascal interface units. The default is on. Don't use with -no_generate_interface_units.
-generate_skeleton_units	Create unit files for Pascal skeleton classes. The default is on. Don't use with -no_generate_skeleton_units.
-generate_stub_units	Create unit files for Pascal stub classes. The default is on. Do not use with -no_generate_stub_units.
-idl_strict	Adhere strictly to the OMG standard interpretation of IDL source. The default is off. Do not use with -no_idl_strict.
-list_files	Lists files written during code generation. The default is off.
-no_comments	Add no comments to generated files. By default, comments are added. Do not use with -comments.
-no_generate_empty_units	Don't create unit files if they are empty. By default, empty units are not created. Don't use with -generate_empty_units.
-no_generate_implementation_units	Don't create Pascal server-side implementation units. By default, the compiler generates implementation units. Do not use with -generate_implementation_units.

Argument	Description
-no_generate_interface_units	Don't create interface units. By default, interface units are created. Don't use with -generate_interface_units.
-no_generate_skeleton_units	Don't create unit files for Pascal skeleton classes. By default, skeleton units are created. Don't use with -generate_skeleton_units.
-no_generate_stub_units	Don't create unit files for Pascal stub classes. By default, stub units are created. Don't use with -generate_stub_units.
-no_idl_strict	Allow extensions to the OMG standard interpretation of IDL source (allowed the default). Do not use with -idl_strict.
-no_overwrite_implementation_units	Don't overwrite implementation units. By default, these are not overwritten. Don't use with -overwrite_implementation_units.
-no_warn_missing_define	Suppress warnings about forward declared names that are not defined. Such warnings are generated by default. Do not use with -warn_missing_define.
-no_warn_unrecognized_pragmas	Suppress warnings about #pragma statements that are not recognized by idl2pas. Such warnings are generated by default. Do not use with -warn_unrecognized_pragmas.
-overwrite_implementation_units	Overwrite existing implementation units. This is off by default. Don't use with -no_overwrite_implementation_units.
-root_dir <path>	Specify the directory where the generated code is written. By default, the code is written to the source idl directory.
-version	Returns the version of the idl2pas compiler. By default, this information is not reported.
-warn_missing_define	Issue a warning when forward declared names are not defined. the default is on. Do not use with -no_warn_missing_define.
-warn_unrecognized_pragmas	Issue a warning if a #pragma statement is not recognized by idl2pas. The default is on. Do not use with -no_warn_unrecognized_pragmas.

idl2ir

This command allows you to populate an interface repository with objects defined in an Interface Definition Language source file.

Syntax idl2ir [-ir <IR_name>] [-replace] {filename.idl}

Example idl2ir -ir my_repository -replace bank/Bank.idl

Description The `idl2ir` command takes an IDL file as input, binds itself to an interface repository server, and populates the repository with the IDL constructs

contained in `infile`. If the repository already contains an item with the same name as an item in the IDL file, the old item is replaced.

Note The `idl2ir` command does not handle anonymous arrays or sequences properly. To work around this problem, `typedefs` must be used for all sequences and arrays.

Option	Description
<code>-ir <name></code>	Specifies the instance name of the interface repository to which <code>idl2ir</code> will attempt to bind. If no name is specified, <code>idl2ir</code> will bind itself to the interface repository server found in the current domain. The current domain is defined by the <code>OSAGENT_PORT</code> environment variable.
<code>-replace</code>	Replaces definitions instead of updating them.
<code>filename.idl</code>	Specifies the IDL file to be used as input.

IDL to Pascal mapping

This chapter describes the basics of VisiBroker for Delphi's current IDL-to-Pascal language mapping, as implemented by the `idl2pas` compiler. VisiBroker for Delphi conforms with the *OMG IDL/Pascal Language Mapping Specification*. A copy of the *OMG IDL/Pascal Language Mapping Specification* is available from the Inprise web site at <http://www.inprise.com/visibroker/>.

See the latest version of the *OMG IDL/Pascal Language Mapping Specification* for complete information and, especially, for information about the following:

- Mapping pseudo-objects to Pascal
- Pascal ORB portability interfaces

This chapter includes the following major sections:

Names	page 2-2
Reserved names	page 2-2
Reserved words	page 2-2
Modules	page 2-3
Basic types	page 2-5
Helper classes	page 2-8
Constants	page 2-9
Constructed types	page 2-10
Interfaces	page 2-14
Mapping for exceptions	page 2-19
User-defined exceptions	page 2-19
System exceptions	page 2-20
Mapping for the Any type	page 2-22
Mapping for Typedef	page 2-23

Names

In general, IDL names and identifiers are mapped to Pascal names and identifiers with no change.

If a name collision might be generated in the mapped Pascal code, the name collision is resolved by prepending an underscore (_) to the mapped name.

In addition, because of the nature of the Pascal language, a single IDL construct may be mapped to several (differently named) Pascal constructs. The “additional” names are constructed by appending a descriptive suffix. For example, the IDL interface `AccountManager` is mapped to the Pascal interface `AccountManager` and the additional Pascal class `TAccountManagerHelper`.

In those exceptional cases that the “additional” names could conflict with other mapped IDL names, the resolution rule described above is applied to the other mapped IDL names. In other words, the naming and use of required “additional” names takes precedence.

IDL names that would normally be mapped unchanged to Pascal identifiers that conflict with Pascal reserved words will have the collision rule applied.

Reserved names

The mapping reserves the use of several names for its own purposes. The use of any of these names for a user-defined IDL type or interface (assuming it is also a legal IDL name) will result in the mapped name having an underscore (_) prepended. Reserved names are as follows:

- The Pascal class `<type>Helper`, where `<type>` is the name of an IDL user-defined type.
- The Pascal class `<type>Helper`, where `<type>` is the name of an IDL user-defined interface.

Reserved words

The use of any of these keywords or constants for a user-defined IDL type or interface (assuming it is also a legal IDL name) will result in the mapped name having an (_) prepended. The reserved keywords in the Pascal language (from the Pascal Language Specification) are as follows:

AND	ARRAY	AS	ASM
BEGIN	CASE	CLASS	CONST
CONST	CONSTRUCTOR	DESTRUCTOR	DISPINTERFACE
DIV	DO	DOWNT0	ELSE

END	EXCEPT	EXPORTS	FILE
FINALIZATION	FINALLY	FOR	FUNCTION
GOTO	IF	IMPLEMENTATION	IN
INHERITED	INITIALIZATION	INLINE	INTERFACE
IS	LABEL	LIBRARY	MOD
NIL	NOT	OBJECT	OF
OR	PACKED	PROCEDURE	PROGRAM
PROPERTY	RAISE	RECORD	REPEAT
RESOURCESTRING	SET	SHL	SHR
STRING	THEN	THREADVAR	TO
TRY	TYPE	UNIT	UNTIL
USES	VAR	WHILE	WITH
XOR			

The reserved Pascal language constants are as follows:

Boolean	Char	WideChar	Shortint
Smallint	Integer	Byte	Word
Extended	Pointer	AnsiChar	Longint
Cardinal	Single	Double	Extended
Real	Real48	Comp	Currency
ShortString	Openstring	file	Text
TextFile	PAnsiChar	PChar	PWideChar
Bytebool	Wordbool	Longbool	AnsiString
WideString	TVarArrayBound	TVarArray	PVarArray
TVarData	PVarData	PShortString	PAnsiString
PWideString	PString	PExtended	PCurrency
TDateTime	PVarRec	String	Variant
TObject	TClass	IUnknown	TInterfacedObject
TGUID	PGUID	Int64	Application
Screen	Print	Printer	

Modules

An IDL module is mapped to a Pascal unit with the same name. All IDL type declarations within the module are mapped to corresponding Pascal class or interface declarations within the generated unit.

IDL declarations not enclosed in any modules are mapped into the (unnamed) Pascal global scope.

Note Cross-module references are mapped differently. See the section on “Interfaces” on page 2-14 for more information.

Code sample 2.1 shows the Pascal code generated for a module declared in an IDL file.

Code sample 2.1 Mapping an IDL module to a Pascal unit

```
//IDL
module Example {...};

{Generated Pascal - interface unit. }
unit Example_i;
...

{Generated Pascal - client (stub) unit. }
unit Example_c;
...

{Generated Pascal - server (skeleton) unit. }
unit Example_s;
...

{Generated Pascal - implementaiton (server side) unit. }
unit Example_Impl;
...
```

Nested Modules

Nested IDL modules are mapped to a multiple file structure in Pascal. This is done because Pascal does not support nested units. An IDL module that contains other IDL modules is mapped to multiple Pascal files. The names of the multiple Pascal units or files are cascaded names derived from the names of the nested modules. These cascaded names start with the outer most module name, and each module name is separated by an underscore (“_”).

For example, an IDL module `m1` that contained a nested module `m2` would generate two Pascal files—a unit named `m1` and a unit named `_m1_m2`. If the IDL nested module `m2` contained an additional nested module `m3`, then it would generate a third Pascal file, and this unit would be named `_m1_m2_m3`.

Code sample 2.2 Mapping nested IDL modules to a Pascal unit

```
// IDL
module m1 {
  module m2 {
    module m3 {
      ...
    };
  };
};
```

```
{Generated Pascal - interface, client(stub), server, and implementation units.}
unit _m1_m2_m3_i;
...

unit _m1_m2_m3_c;
...
unit _m1_m2_m3_s;
...

unit _m1_m2_m3_Impl;
...

unit _m1_m2_i;
...

unit _m1_m2_c;
...

unit _m1_m2_s;
...

unit _m1_m2_Impl;
...

unit m1_i;
...

unit m1_c;
...

unit m1_s;
...

unit m1_Impl;
...
```

By default units are omitted if they are empty. In code sample 2.2 units `m1_i`, `m1_c`, `m1_s`, `m1_Impl`, `m1_m2_i`, `m1_m2_c`, `m1_m2_s`, and `m1_m2_Impl` would be omitted.

Basic types

The following table shows how the defined IDL types map to basic Pascal types.

Table 2.1 Basic type mappings

IDL type	Pascal type
boolean	Boolean
char	Char
wchar	WideChar
octet	Byte
string	AnsiString
wstring	WideString
short	SmallInt

Table 2.1 Basic type mappings (continued)

IDL type	Pascal type
unsigned short	Word
long	Integer
unsigned long	Cardinal
long long	Int64
unsigned long long	Int64
float	Single
double	Double
long double	Extended
fixed ¹	(Not implemented)

1. There is no corresponding type in Pascal.

When there is a potential mismatch between an IDL type and its mapped Pascal type, a standard CORBA exception can be raised. Typically, an exception occurs when the range of the Pascal type is larger than the IDL type. The value must be effectively checked at runtime when it is unmarshaled. This occurs when the value is an `in` parameter, or when it is input for an `inout` parameter. For example, Pascal `Int64` values are a subset of IDL `unsigned long long` values.

Additional details are described in the following sections. In particular, refer to the section “Mapping for exceptions” on page 2-19.

IDL type extensions

This section summarizes VisiBroker’s support for IDL type extensions. The first table provides a summary for quick look-ups. This is followed by a table summarizing support for new types.

Table 2.2 Summary of supported IDL extensions

Type	Supported in VisiBroker for Pascal?
long long	yes
unsigned long long	yes
wchar	yes
wstring	yes
fixed	no ¹

1. There is no corresponding type in Pascal.

Table 2.3 IDL extensions for new types

New types	Description
long long	64-bit signed 2’s complements integers
unsigned long long	64-bit unsigned 2’s complements integers
wchar	Wide characters

Table 2.3 IDL extensions for new types (continued)

New types	Description
wstring	Wide strings
fixed	Fixed-point decimal arithmetic (31 significant digits)

Boolean

The IDL type `boolean` is mapped to the Pascal type `boolean`. The IDL `boolean` constants `TRUE` and `FALSE` are mapped to the corresponding Pascal `boolean` literals `True` and `False`.

Char

Both the IDL and Pascal characters are 8-bit quantities representing elements of a character set. To enforce type-safety, the Java CORBA runtime asserts range validity of all Java `chars` mapped from IDL `chars` when parameters are marshaled during method invocation. If the `char` falls outside the range defined by the character set, a `CORBA::DATA_CONVERSION` exception is thrown.

The IDL `wchar` maps to the Pascal `WideChar` primitive type.

Octet

The IDL type `octet`, an 8-bit quantity, is mapped to the Pascal type `byte`.

String

The IDL type `string`, including both bounded and unbounded variants, is mapped to the Pascal type `AnsiString`. For all strings, range checking for characters in the string is done at marshal time. Range violations cause a `CORBA::DATA_CONVERSION` exception to be raised.

WString

The IDL `wstring` type, including both bounded and unbounded variants, is mapped to the Pascal type `WideString`.

Integer types

The IDL integer types map as shown in Table 2.1, “Basic type mappings,” on page 2-5. That is, the IDL type `short` maps to the Pascal type `SmallInt` and the IDL type `unsigned short` maps to the Pascal type `Word`. The IDL type `long` maps to Pascal `Integer` and the IDL type `unsigned long` maps to the Pascal `Cardinal`. Lastly, the IDL types `long long` and `unsigned long long` map to the Pascal type `Int64`.

There is a potential type mismatch between IDL unsigned long long and the Pascal type `Int64`. The mismatch occurs when an unsigned long long exceeds the range of Pascal `Int64`. When this occurs, the exception `CORBA::DATA_CONVERSION` is thrown.

Floating point types

The IDL floating point type `float` maps to the Pascal type `Single`. The IDL type `double` maps to the Pascal type `Double`. (See Table 2.1, “Basic type mappings,” on page 2-5.)

Long Double types

The IDL type `long double` maps to the Pascal type `Extended`.

Helper classes

All user-defined IDL types have an additional “helper” Pascal class with the suffix `Helper` appended to the type name generated. In general, helper classes support parameter passing for non-simple data types, such as `structs`, `unions`, and user-defined data types. In this context “user defined” includes IDL types defined in various OMG specifications, such as the specifications for the Interface Repository and other OMG services, among others.

Several static methods are supplied for manipulating the type. These include:

- `Any` insert and extract operations for the type
- Getting the repository id
- Getting the typecode
- Reading and writing the type from and to a stream

For any user-defined, non-value type IDL type, `<typename>`, the following is the Pascal code generated for the type. In addition, both the helper class associated with an abstract IDL interface and the helper class for a mapped IDL interface have a `narrow` operation(s) defined for them.

Code sample 2.3 Helper class: Pascal code generated for user-defined non-value type

```
(generated Pascal helper (non value types))

<typename>Helper =class
public
    class procedure Insert(var _A:CORBA.Any;const Value:<typename>);
    class function Extract(const A:CORBA.Any):<typename> >;
    class function TypeCode:CORBA.TypeCode;
    class function RepositoryId:string;
    class function Read(const Input:CORBA.InputStream):<typename> >;
    class procedure Write(const Output:CORBA.OutputStream;const Value:<typename> >;
    class function Narrow(const Obj:CORBA.CORBAObject;IsA: Boolean =False):<typename> >;
    class function Bind(const InstanceName:string ='';HostName:string =''):<typename> >;
end;
```


Code sample 2.4 Mapping of a named type to Pascal helper class

```
//IDL -named type
struct Foo {
    long f1;
    string f2;
};

{generated Pascal helper }
FooHelper =class
public
    class procedure Insert(var _A:CORBA.Any;const Value:Foo);
    class function Extract(const A:CORBA.Any):Foo;
    class function TypeCode:CORBA.TypeCode;
    class function RepositoryId:string;
    class function Read(const Input:CORBA.InputStream):Foo;
    class procedure Write(const Output:CORBA.OutputStream;const Value:Foo);
end;
```

Constants

Constants are mapped to a Pascal constant.

Constants within an interface

IDL constants declared inside an interface are mapped to a Pascal interface constant. The Pascal constant name is formed by concatenating the interface name, prefixed with an underscore, then the name of the constant also prefixed with an underscore: <“_”*interface name*”_”*constant name*>.

Code sample 2.5 Mapping an IDL constant within a module to a Pascal constant

```
//IDL
module Example {
    const long one =1;
    interface Face {
        const long two =2;
    };
};

{Generated Pascal}
unit Example_i;
interface
uses Corba;
const
    one : Integer = 1;
    Face_two : Integer = 2;
```

Constants NOT within an interface

Constants declared in an IDL module are mapped to a Pascal constant with the same name.

Code sample 2.6 Mapping an IDL constant within a module to a Pascal constant

```
// IDL
module Example {
    const long one =1;
};

{generated Pascal }
unit Example_i;
interface
    const one : Integer =1;
```

Constructed types

IDL constructed types include `enum`, `struct`, `union`, `sequence`, and `array`. The types `sequence` and `array` are both mapped to the Pascal `array` type. The IDL constructed types `enum`, `struct`, and `union` are mapped to a Pascal class that implements the semantics of the IDL type. The Pascal class generated has the same name as the original IDL type.

Enum

An IDL `enum` is mapped to a Pascal `enum`. An example follows.

Code sample 2.7 IDL enum mapped to a Pascal enum

```
// IDL
enum EnumType {a, b, c};
{ generated Pascal }
EnumType = ( a, b, c );
```

Struct

An IDL `struct` is mapped simultaneously to both a Pascal interface and a Pascal class. The more natural mapping to a Pascal record cannot be taken because it is legal in IDL to define recursive data structures using `struct` and `sequence`.

Each field of the `struct` is mapped to a property, and that property has both an accessor method and a modifier method defined for it. The accessor method name is the field name with “_get_” prepended to it. The modifier method name is the field name with “_set_” prepended to it. The property name is the same name as the field name in the `struct`.

The Tree example that follows demonstrates how a `struct` maps to a Pascal interface and class.

Code sample 2.8 Mapping an IDL struct to Pascal

```
// IDL
struct StructType {
    long field1;
    string field2;
};
```

```

struct Tree {
    string value;
    sequence <Tree>children;
};

{generated Pascal - _i file}
type StructType = interface;
    Tree      = interface;

{generated Pascal - _c file}
StructType = interface
    function _get_field1:Integer;
    function _get_field2:AnsiString;
    procedure _set_field1(const Value:Integer);
    procedure _set_field2(const Value:AnsiString);
    property field1:Integer read _get_field1
    write _set_field1;
    property field2:AnsiString read _get_field2
    write _set_field2;
end;

TStructTypeStub =class(TInterfacedObject,StructType)
private
    field1:Integer;
    field2:AnsiString;
public
    function _get_field1:Integer;
    function _get_field2:AnsiString;
    procedure _set_field1(const Value:Integer);
    procedure _set_field2(const Value:AnsiString);
end;

_Tree_children =array of Tree;
Tree =interface
public
    function _get_value:AnsiString;
    function _get_children:_Tree_children;
    procedure _set_value(const Value:string);
    procedure _set_children(const Value:_Tree_children);
    property value:AnsiString read _get_value
    write _set_value;
    property children:_Tree_children read _get_children
    write _set_children;
end;

TTreeStub =class(TInterfacedObject,Tree )
private
    value:AnsiString;
    children:_Tree_children;
public
    function _get_value:AnsiString;
    function _get_children:_Tree_children;
    procedure _set_value(const Value:string);
    procedure _set_children(const Value:_Tree_children);
end;

```

Union

An IDL `union` is mapped to a Pascal class. The class provides the following:

- Accessor method for the union's discriminator, which is named `_discriminator`
- Modifier method for the union's discriminator
- Accessor method for each branch
- Modifier method for each branch

If there is a name clash with the mapped union type name or any of the field names, the normal name conflict resolution rule is used: prepend an underscore for the discriminator.

The branch accessor and modifier methods are named after the branch.

Accessor methods shall raise the `CORBA::BAD_OPERATION` system exception if the expected branch has not been set.

Code sample 2.9 Mapping an IDL union to Pascal

```
// IDL
union un_ex switch(long)
{ case 1:long X;
  case 2:string Y;
  case 3:st_ex Z;
};

un_ex = interface
    function _get__discriminator:Integer;
    function _get_X:Integer;
    function _get_Y:AnsiString;
    function _get_Z:st_ex;
    procedure _set__discriminator(const Value:Integer);
    procedure _set_X(const Value:Integer);
    procedure _set_Y(const Value:AnsiString);
    procedure _set_Z(const Value:st_ex);
    property _discriminator:Integer
        read _get__discriminator write _set__discriminator;
    property X:Integer read _get_X write _set_X;
    property Y:AnsiString read _get_Y write _set_Y;
    property Z:st_ex read _get_Z write _set_Z;
end;

un_exImpl =class(un_ex)
private
    F_Disc:Integer;
    FX:Integer;
    FY:AnsiString;
    FZ:sy_ex;
public
    function _get__discriminator:Integer;
    function _get_X:Integer;
    function _get_Y:AnsiString;
    function _get_Z:st_ex;
```

```

    procedure _set__discriminator(const Value:Integer);
    procedure _set_X(const Value:Integer);
    procedure _set_Y(const Value:AnsiString);
    procedure _set_Z(const Value:st_ex);
end;

```

Sequence

An IDL `sequence` is mapped to an unbounded Pascal array with the same name. In the mapping, anywhere the `sequence` type is needed, an array of the mapped type of the sequence element is used.

Bounds checking is done on bounded sequences when they are marshaled as parameters to IDL operations. An error raises an IDL `CORBA::MARSHAL` exception.

Code sample 2.10 Mapping an IDL sequence to PAscal

```

// IDL
typedef sequence<long >   UnboundedSeq;
typedef sequence<long,42 >ShortBoundedSeq;

{generated Pascal - _i file}
type UnboundedSeq = array of Integer;
ShortBoundedSeq  = array of Integer;

```

Code sample 2.11 Mapping of a typedef sequence to Pascal helper class

```

// IDL - typedef sequence
typedef sequence <long>IntSeq;

{generated Pascal helper }
IntSeqHelper =class
public
    class procedure Insert(const A:CORBA.Any;const Value:IntSeq);
    class function Extract(const A:CORBA.Any):IntSeq;
    class function TypeCode:CORBA.TypeCode;
    class function RepositoryId:string;
    class function Read(const Input:CORBA.InputStream):IntSeq;
    class procedure Write(const Output:CORBA.OutputStream;const Value:IntSeq);
end;

```

Array

An IDL array is mapped the same way as an IDL bounded sequence. In the mapping, anywhere the array type is needed, an array of the mapped type of array element is used. In Pascal, the natural Pascal subscripting operator is applied to the mapped array. The bounds for the array are checked when the array is marshaled as an argument to an IDL operation and a `CORBA::MARSHAL` exception is raised if a bounds violation occurs. The length of the array can be made available in Pascal by bounding the array with an IDL constant, which will be mapped as per the rules for constants.

Code sample 2.12 Mapping for an array

```
// IDL
const long ArrayBound = 42;
typedef long LongArray [ArrayBound];

{generated Pascal }
const ArrayBound : Integer = 42;
type LongArray =array [0..41 ] of Integer;
```

Interfaces

IDL interfaces are mapped to two public Pascal interfaces—a signature interface and an operations interface. The signature interface extends `IDLEntity` and has the same name as the IDL interface name. It is used as the signature type in procedure and function declarations when interfaces of the specified type are used in other interfaces. The operations interface has the same name as the IDL interface with the suffix `Operations` appended to its name. It is used in the server-side mapping and as a mechanism for providing optimized calls for co-located clients and servers. An additional “helper” Pascal interface or class with the suffix `Helper` appended to the IDL interface name is also generated. A stub class is also generated, and this class is explained below.

The helper class holds a static narrow method that allows an instance of `CORBA.Object` to be narrowed to the object reference of a more specific type. The IDL exception `CORBA:BAD_PARAM` is thrown if the narrow fails.

There are no special “null” object references. Pascal `nil` can be passed freely wherever an object reference is expected.

Attributes are mapped to a pair of Pascal methods—an accessor and a modifier method—and to a property declaration. The accessor function name is the name of the attribute with “_get_” prepended. The modifier procedure is the name of the attribute with “_set_” prepended. The property declaration name is the same as the attribute with the addition of read and write clauses that refer to the accessor and modifier methods respectively. If the attribute is readonly, only the accessor method is generated and the property declaration only has a read clause.

Abstract Interfaces

Abstract interfaces are mapped to Pascal operations interfaces in the same way as regular IDL interfaces, with the exception that the mapped class has the same name as the IDL interface.

Helper classes are generated in the usual way.

Code sample 2.13 Mapping an IDL interface to Pascal

```
//IDL
module example {
    interface MyInterface {
        long op(in string s );
    };
};
```

```

};

{generated Pascal }
unit example_i;
interface
type MyInterface = interface
    function op(const s:AnsiString) : Integer;
end;

```

Multiple Inheritance

VisiBroker for Pascal does not support multiple inheritance of interfaces. Thus, multiple inheritance relationships between IDL interfaces cannot be expressed directly. Instead, single inheritance is applied to one interface (in lexicographical order) and the operations of the other inherited interfaces are copied into the derived interface.

There are generated IDL stub classes that, in turn, generate Pascal code.

Code sample 2.14 Generated IDL stub classes

```

//IDL
module M {
    interface A {
        void A1();
        void A2();
    };
    interface B {
        void B1();
        void B2();
    };
    interface AB : B,A {
        void AB1();
        void AB2();
    };
};

```

Code sample 2.15 Generated Pascal code

```

unit M_i;
interface
uses CORBA;
type A = interface;
    B = interface;
    AB = interface;

A = interface
    procedure A1;
    procedure A2;
end;

B = interface
    procedure B1;
    procedure B2;
end;

```

```
AB = interface(A)
  procedure B1;
  procedure B2;
  procedure AB1;
  procedure AB2;
end;

TASStub =class(TStub,A)
public
  procedure A1;
  procedure A2;
end;

TBStub =class(TStub,B)
public
  procedure B1;
  procedure B2;
end;

TABStub =class(TASStub, AB ,B)
public
  procedure B1;
  procedure B2;
  procedure AB1;
  procedure AB2;
end;

TASkeleton =class(TSkeleton, A)
public
  procedure A;
  procedure B;
end;

TBSkeleton =class(TSkeleton,B)
public
  procedure B1;
  procedure B2;
end;

TABSkeleton =class(TASkeleton, AB ,B)
public
  procedure B1;
  procedure B2;
  procedure AB1;
  procedure AB2;
end;
```

Cross-Module Referencing

VisiBroker for Pascal does not support recursive referencing across unit boundaries, though in IDL such cases can be legally specified, such as by using nested modules or re-opening modules.

For these cases, the mapping breaks the rules defined earlier for the mapping of modules and interfaces. Instead, a single unit is created and named after the first module. Interfaces defined in nested modules are prefixed with an

underscore followed by the module name and another underscore, as follows:
 “_”<module name>”_”.

The following examples illustrate recursive referencing for nested modules and re-opened modules.

Code sample 2.16 Nested modules

```
//IDL - Cross-referenced nested modules
module m1 {
  interface if1;
  module m2 {
    interface if2 {
      m1::if1 getIf1();
    };
    interface if1 {
      m2::if2 getIf2();
    };
  };
};

unit m1_i;
interface
uses CORBA;
type _m2_if2 = interface;
  if1 = interface;
    _m2_if2 = interface
      function getIf1:If1;
    end;

  if1 = interface
    function ifIf2:_m2_if2;
  end;
....
end.
```

Code sample 2.17 Re-opened modules

```
//IDL - re-opened modules
module m1 {
    interface if1;
};

module m2 {
    interface if2 {
        m1:if1 getIf1();
    };
};

module m1 {
    interface if1 {
        m2:if2 getIf2();
    };
};

unit m1_i;
interface
uses CORBA;
type _m2_if2 = interface;
    if1 = interface;
        _m2_if2 = interface
            function getIf1:Iif1;
        end;
    if1 =interface
    function ifIf2:_m2_if2;
end;
....
end.
```

Passing parameters

IDL *in* parameters, which implement call-by-value semantics, are mapped to “const” Pascal parameters. IDL *out* parameters, which implement call-by-result semantics, are mapped to “out” Pascal parameters. IDL *inout* parameters, which implement call-by-value semantics, are mapped to “var” Pascal parameters.

IDL operations with a void operation result are mapped to Pascal procedures. All other IDL operations are mapped to Pascal functions. The results of IDL operations are returned as the result of the corresponding Pascal function.

Code sample 2.18 Parameter mapping

```
//IDL
module Example {
    interface Modes {
        long operation(in    long inArg,
                      out    long outArg,
                      inout long inoutArg);
    };
};
```

```

{Generated Pascal }
unit Example;
interface
type
Modes =interface
    function operation (const inArg:Integer; out outArg:Integer; var inoutArg:Integer);
end;

```

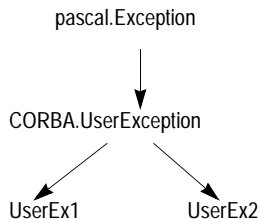
Context Arguments to Operations

When an IDL operation has a context specification, a `CORBA.Context` input parameter is appended to the invocation argument list following the operation-specific arguments.

Mapping for exceptions

IDL exceptions are mapped to Pascal exceptions. Pascal exceptions are represented by a Pascal class which inherits from the base class `Exception`. This mapped Pascal class provides instance variables for the exception fields and for constructors.

User defined exceptions inherit (indirectly) from the Pascal `Exception` class via the class `CORBA.UserException`.



User-defined exceptions

User-defined exceptions are mapped to Pascal classes that extend `CORBA.UserException`.

If the exception is defined within a nested IDL scope (essentially within an interface) then its Pascal class name is prefixed with a leading underscore, the interface name, followed by another underscore: `<“_”interface name”_”>`.

The definition of the Pascal class is as follows:

```

{ Pascal }
unit CORBA;
type
    UserException = class(Exception)
        constructor Create(const Value: AnsiString = '');
    end;

```

Code sample 2.19 Mapping user-defined exceptions

```
//IDL
module Example {
    exception ex1 {long reason_code;};
    interface Face {
        exception ex2{string msg;};
    };

    {Generated Pascal }
    unit Example;
    type

    ex1 =class(CORBA.UserException)
    public
        reason_code:Integer;
    end;

    _Face_ex2 =class(CORBA.UserException)
    public
        msg:AnsiString;
    end;
```

Unknown user exceptions

VisiBroker for Pascal has one standard user exception, referred to as the unknown user exception. Because the ORB does not know how to create user exceptions, it wraps the user exception as an `UnknownUserException` and passes it out to the DII layer. The exception is specified as follows:

```
unit CORBA;

type
    UnknownUserException = class(CORBA.UserException)
    public
        except: Any;
    end;
```

In addition, there are several PIDL exceptions which are also mapped into user exceptions. See Section, “Certain Exceptions,” for more details.

System exceptions

The standard IDL system exceptions are mapped to final Pascal classes that extend `CORBA.SystemException` and provide access to the IDL major and minor exception code, as well as a string describing the reason for the exception. There are no public constructors for `CORBA.SystemException`; only classes that extend it can be instantiated.

The Pascal class name for each standard IDL exception is the same as its IDL name and is declared to be in the `CORBA` package. The default constructor supplies 0 for the minor code, `COMPLETED_NO` for the completion code, and the empty string (“”) for the reason string. There is also a constructor which takes

the reason and uses defaults for the other fields, as well as one which requires all three parameters to be specified.

Table 2.4 IDL System Exception to Pascal Class Name Mapping

IDL Exception Name	Pascal Class
CORBA::UN_KNOWN	CORBA.UNKNOWN
CORBA::BAD_PARAM	CORBA.BAD_PARAM
CORBA::NO_MEMORY	CORBA.NO_MEMORY
CORBA::IMP_LIMIT	CORBA.IMP_LIMIT
CORBA::COMM_FAILURE	CORBA.COMM_FAILURE
CORBA::INV_OBJREF	CORBA.INV_OBJREF
CORBA::NO_PERMISSION	CORBA.NO_PERMISSION
CORBA::INTERNAL	CORBA.INTERNAL
CORBA::MARSHAL	CORBA.MARSHAL
CORBA::INITIALIZE	CORBA.INITIALIZE
CORBA::NO_IMPLEMENT	CORBA.NO_IMPLEMENT
CORBA::BAD_TYPECODE	CORBA.BAD_TYPECODE
CORBA::BAD_OPERATION	CORBA.BAD_OPERATION
CORBA::NO_RESOURCES	CORBA.NO_RESOURCES
CORBA::NO_RESPONSE	CORBA.NO_RESPONSE
CORBA::PERSIST_STORE	CORBA.PERSIST_STORE
CORBA::BAD_INV_ORDER	CORBA.BAD_INV_ORDER
CORBA::TRANSIENT	CORBA.TRANSIENT
CORBA::FREE_MEM	CORBA.FREE_MEM
CORBA::INV_IDENT	CORBA.INV_IDENT
CORBA::INV_FLAG	CORBA.INV_FLAG
CORBA::INTF_REPOS	CORBA.INTF_REPOS
CORBA::BAD_CONTEXT	CORBA.BAD_CONTEXT
CORBA::OBJ_ADAPTER	CORBA.OBJ_ADAPTER
CORBA::DATA_CONVERSION	CORBA.DATA_CONVERSION
CORBA::OBJECT_NOT_EXIST	CORBA.OBJECT_NOT_EXIST
CORBA::TRANSACTION_REQUIRED	CORBA.TRANSACTION_REQUIRED
CORBA::TRANSACTION_ROLLEDBACK	CORBA.TRANSACTION_ROLLEDBACK
CORBA::INVALID_TRANSACTION	CORBA.INVALID_TRANSACTION

There is a similar definition for each Pascal exception class that maps to a standard IDL system exception. A typical definition of the Pascal exception class is as follows.

```
{ from CORBA unit }
unit CORBA;
type
  CompletionStatus = (YES, NO, MAYBE);
  SystemException = class(Exception)
  public
    minor: Integer;
    completed: CompletionStatus;
  end;

  UNKNOWN = class(CORBA.SystemException);
{ ... }
```

Mapping for the Any type

The IDL type `Any` maps to the Pascal type `Variant`. There are two classifications of `Any`s in VisiBroker for Delphi: simple types and complex types. Simple types include IDL types such as shorts, longs, floats, and so on. Complex types are built from simple types. They include IDL types such as structs, sequences, arrays, interface objects, and so on.

A simple type is assigned to an `Any` by an assignment statement. This is identical to how simple types are mapped to `Variants` in Object Pascal. For example, an IDL short integer can be assigned to an `Any` as follows:

```
procedure Test;
var MyAny: Any;
begin
  MyAny := 123;
end;
```

Complex types have an associated helper class that must be used for handling the `Any`. This helper class has two methods, `Insert` and `Extract`.

The `Insert` method takes an `Any` and a value of its complex type as arguments. It assigns the value to the `Any`. For example, the `Insert` method of a struct helper looks like the following:

```
class procedure TMyStructHelper.Insert(var _A: Corba.Any;
                                       const Value: MyStruct_i.MyStructType);
begin
  ...
end;
```

The `Extract` method takes an `Any` as an argument and returns its value as the complex type. For example, the `Extract` method of a struct helper looks like the following:

```
class function TMyStructHelper.Extract(const _A: Corba.Any): MyStruct_i.MyStructType;
begin
  ...
end;
```

Mapping for Typedef

The IDL `Typedef` is mapped to the Pascal `Typedef` and type declarations.

IDL sequence definitions are mapped to a Pascal type declaration. All other IDL typedefs are mapped to Pascal typedefs.

Code sample 2.20 Mapping IDL `Typedef` to Pascal

```
// IDL
struct EmpName {
    string firstName;
    string lastName;
};
typedef EmpName EmpRec;
typedef sequence <long> IntSeq;

{ generated Pascal }

EmpName = record
    firstName: AnsiString;
    lastName: AnsiString;
end;

EmpRec = type EmpName;

IntSeq = array of Integer;
```


Index

Symbols

-? argument 1-2

A

arguments

- ? 1-2
- C 1-2
- comments 1-2
- D 1-2
- define 1-2
- generate_empty_units 1-2
- generate_implementation_units 1-2
- generate_interface_units 1-2
- generate_skeleton_units 1-2, 1-3
- generate_stub_units 1-2
- H 1-2
- h 1-2
- help 1-2
- I 1-2
- idl_strict 1-2
- include 1-2
- ir 1-4
- list_files 1-2
- list_includes 1-2
- no_comments 1-2
- no_generate_empty_units 1-2
- no_generate_implementation_units 1-2
- no_generate_interface_units 1-3
- no_generate_stub_units 1-3
- no_idl_strict 1-3
- no_line_directives 1-2
- no_overwrite_implementation_units 1-3
- no_warn_missing_define 1-3
- no_warn_unrecognized_pragmas 1-3
- overwrite_implementation_units 1-3
- P 1-2
- replace 1-4
- retain_comments 1-2
- root_dir 1-3
- U 1-2
- undefine 1-2
- usage 1-2
- version 1-3
- warn_missing_define 1-3
- warn_unrecognized_pragmas 1-3

arrays

mapping 2-10

B

boolean type
mapping 2-7

C

-C argument 1-2
char type
mapping 2-7
commands
idl2ir 1-3
-comments argument 1-2
constants
mapping 2-9
constructed types
mapping 2-10

D

-D argument 1-2
-define argument 1-2

E

enums
mapping 2-10
exceptions
mapping 2-19

F

floating point
mapping 2-8

G

-generate_empty_units argument 1-2
-generate_implementation_units argument 1-2
-generate_interface_units argument 1-2
-generate_skeleton_units argument 1-2, 1-3
-generate_stub_units argument 1-2
Generating pascal code 1-1

H

-H argument 1-2
-h argument 1-2
-help argument 1-2
Helper classes
mapping 2-8

I

-I argument 1-2

IDL

- mapping constants 2-9
- mapping constructed types 2-10
- mapping interfaces 2-14
- mapping modules 2-3
- mapping names to Java 2-2
- mapping parameters 2-18
- mapping to Java 2-1
- mapping types 2-5
- reserved names 2-2
- reserved words 2-2
- type extensions 2-6

IDL compiler 1-1

-idl_strict argument 1-2

idl2ir

- command info 1-3
- description 1-3

idl2ir

- ir 1-4
- replace 1-4

idl2pas compiler

- ? 1-2
- C 1-2
- comments 1-2
- D 1-2
- define 1-2
- generate_empty_units 1-2
- generate_implementation_units 1-2
- generate_interface_units 1-2
- generate_skeleton_units 1-2, 1-3
- generate_stub_units 1-2
- H 1-2
- h 1-2
- help 1-2
- I 1-2
- idl_strict 1-2
- include 1-2
- list_files 1-2
- list_includes 1-2
- no_comments 1-2
- no_generate_empty_units 1-2
- no_generate_implementation_units 1-2
- no_generate_interface_units 1-3
- no_generate_stub_units 1-3
- no_idl_strict 1-3
- no_line_directives 1-2
- no_overwrite_implementation_units 1-3
- no_warn_missing_define 1-3
- no_warn_unrecognized_pragmas 1-3
- overwrite_implementation_units 1-3
- P 1-2
- retain_comments 1-2

-root_dir 1-3

-U 1-2

-undefine 1-2

-usage 1-2

-version 1-3

-warn_missing_define 1-3

-warn_unrecognized_pragmas 1-3

idl2pas tool 1-1

-include argument 1-2

Interface Repository

populating with idl2ir 1-3

interfaces

mapping 2-14

-ir argument 1-4

J

Java

mapping from IDL 2-1

L

-list_files argument 1-2

-list_includes argument 1-2

M

mapping

- arrays 2-10
- boolean type 2-7
- char type 2-7
- constants 2-9
- constructed types 2-10
- enums 2-10
- exceptions 2-19
- floating point 2-8
- Helper classes 2-8
- IDL names 2-2
- IDL types 2-5
- interfaces 2-14
- modules 2-3
- octet 2-7
- passing parameters 2-18
- reserved names 2-2
- reserved words 2-2
- sequences 2-10
- string 2-7
- structs 2-10
- unions 2-10

modules

mapping 2-3

N

-no_comments argument 1-2

-no_generate_empty_units argument 1-2

- no_generate_implementation_units argument 1-2
- no_generate_interface_units argument 1-3
- no_generate_stub_units argument 1-3
- no_idl_strict argument 1-3
- no_line_directives argument 1-2
- no_overwrite_implementation_units argument 1-3
- no_warn_missing_define argument 1-3
- no_warn_unrecognized_pragmas argument 1-3

O

- octet
 - mapping 2-7
- overwrite_implementation_units argument 1-3

P

- P argument 1-2
- parameters
 - mapping 2-18
- Programmer Tools
 - general information 1-1
- Programmer tools
 - idl2pas 1-1

R

- replace argument 1-4
- reserved names
 - mapping 2-2
- reserved words
 - mapping 2-2
- retain_comments argument 1-2
- root_dir argument 1-3

S

- sequences
 - mapping 2-10
- string
 - mapping 2-7
- structs
 - mapping 2-10

T

- tools
 - idl2ir 1-3
- type extensions 2-6
- types
 - mapping 2-5

U

- U argument 1-2
- undefine argument 1-2
- unions
 - mapping 2-10
- usage argument 1-2

V

- version argument 1-3

W

- warn_missing_define argument 1-3
- warn_unrecognized_pragmas argument 1-3
- words
 - reserved 2-2
- wstring
 - mapping 2-7