Babel
Users' Guide

TAMARA DAHLGREN THOMAS EPPERLY
GARY KUMFERT JAMES LEEK

Disclaimer

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Release Information

Babel Users' Guide (this document)	UCRL-SM-205559
Babel Source Code (associated software)	UCRL-CODE-2002-054

Babel
Users' Guide

TAMARA DAHLGREN THOMAS EPPERLY
GARY KUMFERT JAMES LEEK

Center For Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, California, USA

June 27, 2006

Preface

This document applies to Babel 0.99.0. It, like the software it documents, is a work in progress.
– The Babel Development Team

Babel in a Nutshell

Babel is a tool that enables software written in different languages to communicate. It accomplishes this task by using an Interface Definition Language (IDL) similar to COM and CORBA. Babel relies on the Scientific Interface Definition Language (SIDL) that is specifically tuned for scientific applications. By expressing software interfaces, or APIs[1], in SIDL the appropriate glue code stubs and skeletons can be generated to facilitate language interoperability. Features unique to SIDL are:

- Dynamic multi-dimensional arrays
- Complex numbers (e.g. $2 + 3i$)
- In-process optimizations
- Special directives for large-scale parallel distributed programming (future)
- Syntax for specifying interface behavior (future)

Babel enables true object-oriented techniques even in non object-oriented languages. The object model that SIDL supports is similar to Java and Objective C where a class can extend at most one class, but implement many interfaces. In C++ speak, an interface is simply a class of all pure-virtual methods. Furthermore, if library developers want object-oriented features but are required to be 100% ANSI C compliant, Babel can meet those constraints. Although the Babel code generator is implemented in Java, the runtime libraries and generated files for C bindings are 100% ANSI C compliant.

Babel can be used as the basis for a component framework, but it is *not* a complete framework by itself. We've added a tiny CCA-compliant framework, called *Decaf*, in our examples/ directory. Decaf demonstrates how Babel can be used to implement a component framework.

SIDL is also a useful communications tool for code development teams since it only expresses the public API. That is, implementation details, which often prove distracting during collaborative design, can be safely avoided by restricting discussions to the interfaces described in SIDL. Furthermore, since SIDL is simple and clean it can be used by Computer Scientists, Math Programmers, and Application Scientists to debate APIs even using only email.

Scope of this Manual

This document is intended as an introduction and tutorial on the use of Babel tools for the generation and use of component software. The Babel tools were designed specifically for scientific applications, therefore most of the examples and exercises here also deal with scientific applications.

This manual assumes the reader is a programmer who is proficient in two or more of the following languages: C, C++, FORTRAN 77, FORTRAN 90, Java, or Python. Furthermore, this manual assumes the reader is familiar with the

[1] Application Programming Interfaces

SPMD[2] programming model that pervades the scientific computing community. Knowledge of and experience with MPI programming is helpful, but not strictly required.

Getting the Software

Babel source is available free of charge on the web. Developed by the Components Project at the Lawrence Livermore National Laboratory Center for Applied Scientific Computing (CASC), it is licensed under the Lesser GNU Public License (LGPL). See the source distribution for details.

The homepage for the Components Project is

http://www.llnl.gov/CASC/components

Conventions

The following typographic conventions are used throughout this manual.

Italic	is used for file and command names. It is also used to highlight comments in examples and to define terms the first time they appear in a document.
`Constant Width`	is used in examples to show the text that is generated, and in regular text to show operators, variables, and the output from commands or programs.
`Constant Slanted`	is used for displaying for SIDL source code. We use a separate font to distinguish SIDL code from generated code.
`Constant Bold`	is used to show user's modifications to generated code and in examples to show user's actual input at a terminal.
Sans Serif Slanted	is used in examples to show variables for which a context-specific substitution should be made. The variable *filename*, for example, would be replaced by the actual filename.

Additionally, we may use specific blocks of text as sidebars to call the readers attention to particular information. Here's one kind.

> **Rationale:** *Often when listing restrictions or requirements, we find it helpful to also explain and document the rationale behind a design decision. In time, the context in which the rationale was based may become irrelevant, making the rationale blocks very useful for understanding when to change a decision.*

We Appreciate Your Feedback

We have tested and verified the information in this manual. Nonetheless, features may have changed or oversights may exist. Please contact us with any issues, corrections, or suggestions for future versions of this manual through snail mail at:

Components Project
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L–365
Livermore, CA 94551

[2] Single Program Multiple Data

or through email to:

components@llnl.gov

To find out more about Babel, feel free to subscribe to one or more of the associated distribution lists given below.

- babel-announce@llnl.gov is a moderated email forum to which anyone can subscribe (though no-one can post). This is a low-volume alternative for people who want to know about releases and major announcements.
- babel-dev@llnl.gov is an open discussion forum about Babel for serious babel users who want to talk about the internal workings of the tools. Anyone can subscribe or send email to this list.
- babel-users@llnl.gov is an open discussion forum about Babel for users. Anyone can subscribe or send email to this list.

To subscribe, simply send email to majordomo@lists.llnl.gov with the appropriate line(s):

subscribe babel-announce *[email-address]*
subscribe babel-dev *[email-address]*
subscribe babel-users *[email-address]*

where you can explicitly state your email address in *email-address* or, if you leave *email-address* blank, majordomo will use your email ReplyTo: field.

Acknowledgments

Project Alumni: Nathan Dykman, Scott Kohn, and Brent Smolinski

Interns: Melvina Blackgoat, Kirk Kelsey, Sarah Knoop, and Nija Shi

Alpha Testers: Andy Cleary, Jeff Painter, Cal Ribbens

Contributors (Ideas, Bug Reports, Patches, & Code): Rob Armstrong, Ben Allan, Wael Elwasif, Matt Knepley, Boyana Norris, Barry Smith, Jody Winston, and many more.

Sponsors: Babel development originally started as a Strategic Initiative (SI) in the LDRD (Lab Directed R&D) portfolio of Lawrence Livermore National Laboratory.

Current funding is from the DOE/Office of Science SciDAC program as part of the Common Component Technology for Terascale Scientific Simulation (CCTTSS). Also known as the Common Component Architecture.

Software Notices

Babel depends on a great deal of third-party software.

- **JavaCC** is used to generate the SIDL Parser. This is a java.net community project. JavaCC is available under a BSD-style license here: https://javacc.dev.java.net/).
- **gnu.getopt** is an implementation of GNU Getopt in Java and is distributed with Babel as a JAR file. It can be downloaded (along with sourcecode) from either the GNU website

  http://www.gnu.org/software/java/packages.html

  or the author's website

  http://www.urbanophile.com/arenn/hacking/download.html.

The following is the copyright notice for gnu.getopt:

```
/**********************************************************************
/* Getopt.java -- Java port of GNU getopt from glibc 2.0.6
/*
/* Copyright (c) 1987-1997 Free Software Foundation, Inc.
/* Java Port Copyright (c) 1998 by Aaron M. Renn (arenn@urbanophile.com)
/*
/* This program is free software; you can redistribute it and/or modify
/* it under the terms of the GNU Library General Public License as published
/* by  the Free Software Foundation; either version 2 of the License or
/* (at your option) any later version.
/*
/* This program is distributed in the hope that it will be useful, but
/* WITHOUT ANY WARRANTY; without even the implied warranty of
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
/* GNU Library General Public License for more details.
/*
/* You should have received a copy of the GNU Library General Public License
/* along with this program; see the file COPYING.LIB.  If not, write to
/* the Free Software Foundation Inc., 59 Temple Place - Suite 330,
/* Boston, MA  02111-1307 USA
/**********************************************************************/
```

The text for the GNU Library GPL is available at `http://www.gnu.org/copyleft/library.html`.

Contents

Chapter 1

Introduction

Contents

1.1 Babel Facilitates Language Interoperability

Babel was conceived, designed, and built to solve a problem; namely, to make scientific software libraries equally accessible from all of the standard languages. Hence, its goal is language interoperability. The vision goes far beyond calling BLAS[1] implemented in FORTRAN 77 from a C program. At its heart, Babel lets programmers use their tool of choice in developing complete applications using components implemented in one or more distinct programming languages.

For instance, let us say that an application scientist is running a sophisticated C++ code from a Python scripting environment. This can already be easily accomplished with technologies like SWIG. Now let's say that the simulation is showing some erratic behavior and the application scientist wants to extend the `ConvergenceCheck` class to also report some information to a log file. Let's also assume that this application scientist doesn't want to write a new C++ class much less rewrite the current application. What this individual wants to do is derive and utilize a new class in Python from the C++ `ConvergenceCheck` class. Thus, the C++ simulation code will now have to invoke a method on a class implemented in Python, which then dispatches back to the C++ base class after doing its additional logging. This cannot be done in SWIG because SWIG does not support calls from C++ to Python, only from Python to C++. Figure 1.1 shows a high level view of what Babel's solution looks like. The developers write the application in Python, the library in C++, and the extended `ConvergenceCheck` in Python. All the glue code is generated by running Babel on a SIDL file. This is an example of a capability that Babel provides that is outside the scope of SWIG.

Figure 1.2 lists many of the primary languages that are of interest to scientific simulation software developers and users. The good news is that there is a path from each language to every other; meaning that calling from one to another is possible. However, the technologies to get from one language to another vary widely, are fraught with pitfalls, and may require calling through a completely different language.

Babel works by providing the technology to define and support the multi-language interoperation of a common subset of functionality through programming language-neutral interface specifications. See Fig. 1.3 to see a graphical representation of the supported languages. It is important to note that this common functionality subset is *far* from a lowest common denominator solution in that Babel actually adds functionality when it is lacking in the host language.

[1] BLAS: Basic Linear Algebra Subroutines

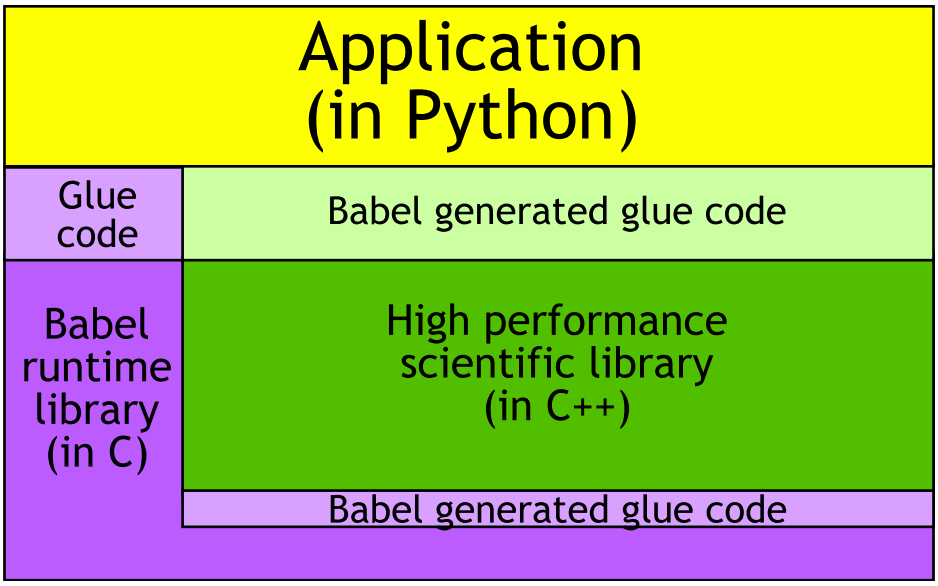

Figure 1.1: Example Babel multi-language application

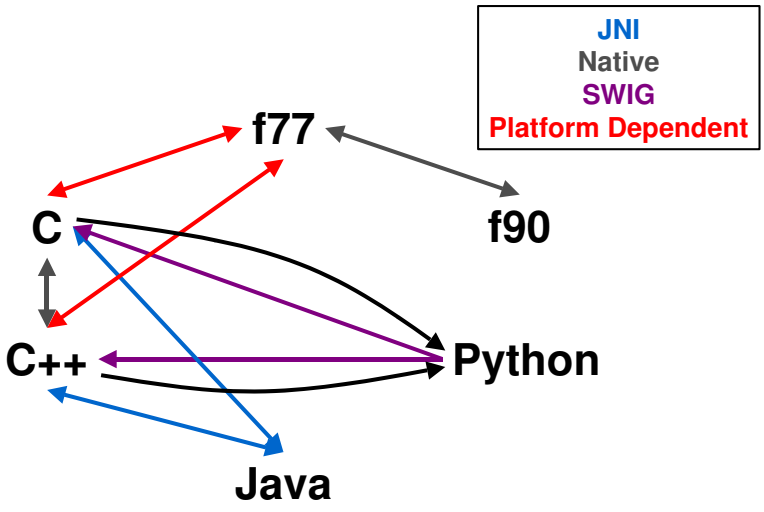

Figure 1.2: Language Interoperability Using Current Technology.

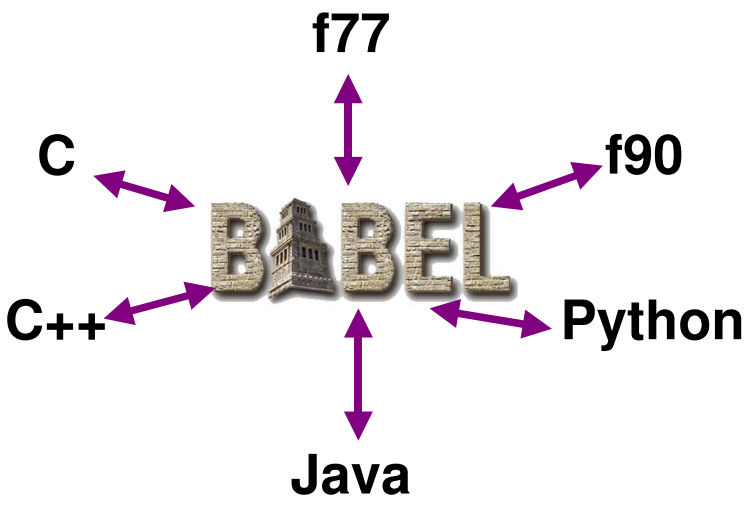

Figure 1.3: Language Interoperability Using Babel.

1.2 Scientific Interface Definition Language (SIDL)

In order to support multi-language interoperability, Babel relies on the specification of interfaces in the Scientific Interface Definition Language (SIDL) (pronounced "SIGH-dull"). SIDL is similar to COM and CORBA IDLs, but was designed with an emphasis on scientific computing. Specifically, SIDL supports dynamic multi-dimensional arrays and has built-in complex numbers. It will acquire a set of directives to aid in the description of massively parallel distributed objects and additional syntax for specifying interface behavior.

When it comes to deciding what programming idioms to support across all languages and which ones to reject, SIDL strikes a careful balance between minimalism and completeness. It is *not* a lowest common denominator solution. SIDL is minimal to keep the learning curve as low as possible. It is complete so developers do not feel constrained in how to express their solutions.

SIDL is object-oriented. Its object model closely resembles that of Java and Objective C. In this model there is single inheritance of implementation and multiple inheritance of interfaces. It supports the typical notions of virtual, static, and final methods. SIDL also provides a basic set of features by defining and implementing the basic types for interfaces, classes and exceptions. All types implicitly inherit from these basic types.

The most important concept to grasp about SIDL is that SIDL only defines a public interface that other programs may use to access your code. As a result, all methods defined as part of a SIDL file are public; if you do not want a method to be globally useable, simply do not define it in your SIDL file. Furthermore, all object and class data is implicitly private. There is no way to declare or define data in a SIDL file. Instead, any data required for your code should be declared in the implementation language files. This way, the languages that use your code through Babel may create your objects and pass them around just like any normal piece of data, but they may only access the data through the provided interface.

SIDL also has a complete set of fundamental data types, from booleans to double precision complex numbers. It also supports more sophisticated types such as enumerations, strings, objects, and dynamic multi-dimensional arrays.

SIDL is still a work in progress. Of particular research interest are directives that will be added for parallel distributed object interaction and features to specify behavioral semantics associated with the interfaces.

1.3 Benefits to Customers

Babel has two types of customers: *developer* and *user*. The developer implements a library that will be used by one or more users, and the user uses libraries via their Babel generated interfaces. Babel provides value to developers by making their software accessible to a larger customer base because their software can be used from more different

programming languages. Users, on the other hand, may not care or even know that they are interacting with a library through Babel. Babel provides value to users by making more software accessible to them.

Babel provides some features that benefits user and developer alike. The most important aspect to note here is that all Babel objects are reference counted. This feature is critical to encapsulate the memory allocation library (e.g. C's malloc/free or C++'s new/delete) used in the implementation of the object. Users never need concern themselves with when to free up a resource, they only declare when they're done with their reference to that resource. Developers are free to use different memory allocation subsystems in different parts of their code if need be.

Babel also provides a consistent type system across all languages. This is particularly important when you want to make object-oriented libraries written in C++, Python or Java accessible in procedural languages like Fortran 77, Fortran 90 and C. Babel maps the object-oriented concepts like inheritance, polymorphism and object identity into all its languages.

1.4 Beyond Babel's Scope

The language interoperability problem is a large one, and though the Babel tools address much of it, there is still a lot that is beyond the scope of our tool. Babel is at its heart a code generator and a runtime library. Consequently, the following features are currently limitations of the Babel tool kit:

Reverse engineering is not supported. That is, there is no support for inspecting or modifying compiled code. In addition, scanning existing software to generate SIDL wrappers is not supported. There are other groups who are pursuing a C++ to SIDL converter. Since SIDL contains different information than what is in a C++ header file, however, such a converter cannot be fully automated without additional help.

Library compatibility is limited. Since Python and Java dynamically load libraries into their virtual machines, using these languages requires the ability to build shared libraries. In general, building shared libraries (particularly from C++) is difficult and error prone. This is compounded by the fact that compiler vendors have no standard way of doing this, and many tools that help building shared libraries don't support C++. One can build a legitimate shared library that still won't work because there are unresolved symbols, or the library was loaded in the wrong mode.

Compiler compatibility is limited. Since the C++ standard does not specify a binary interface and uses a lot of hashing in their symbol tables, there have been no attempts to get libraries from dissimilar C++ compilers to work together. Similarly, although we support FORTRAN 77 and FORTRAN 90, all libraries of Fortran code must be compiled with the same compiler... again because of the lack of a standard binary interface.

Despite the aforementioned limitations, Babel does facilitate the development of language interoperable software. However, issues of robust packaging, building, and deployment of language interoperable software still loom on the horizon.

1.5 Summary

Babel consists of a set of tools that are intended to be used for facilitating language interoperability in the scientific computing community. Using interfaces for libraries or components specified in Scientific Interface Definition Language (SIDL) files, Babel can generate corresponding XML representations as well as the source code for the corresponding stubs, intermediate object representations, and implementation skeletons. The generated source code then becomes the foundation for the glue code that is used for language interoperability between callers of libraries and components.

In addition to providing generated code that automatically handles mapping fundamental data type parameters associated with calls between different languages, Babel has built-in support for complex numbers and multi- dimensional arrays. Additional benefits include object reference counting to facilitate memory management.

Finally, Babel's primary goal is to facilitate the development of language interoperable libraries and components. Hence, support for reverse engineering is not provided. Given that Babel has been developed by a research team, there are also limitations associated with shared library and programming language-specific compiler interoperability support that have been looked into but probably will not be addressed in the foreseeable future. Regardless, Babel

has proven to be useful to its stakeholders to the point that it is becoming an integral part of the Common Component Architecture (CCA). Refer to papers and presentations on our web site for more information.

1.6 Organization

The remainder of this document is separated into two parts; namely, foundations and supported language bindings. Part I is devoted to describing the SIDL and the Babel tools. It starts with a tutorial to gently introduce the reader to the development of glue code from both the implementation (or server) and user (or client) sides. The following chapter introduces SIDL and Babel basics. Finally, a chapter on advanced topics, such as linking options, is provided.

Part II describes the language bindings currently supported by Babel. At this point, most of the bindings are programming languages. In which case, most have both client- and server-side bindings. However, Babel also supports textual language backends. At this time, Extensible Markup Language (XML), Hypertext Markup Language (HTML) and Scientific Interface Definition Language (SIDL) are the only textual backends that are supported.

Appendices are included to provide more information on topics such as acronyms, the SIDL Grammar, and SIDL XML. In addition, sections are included that provide advice and tips on troubleshooting.

Part I

Foundations

Chapter 2

Installation

Ideally, Babel will configure and make "out-of-the-box" on most Unix-like machines. If the configuration process detects that certain resources are unavailable, it will correctly disable support for languages or features needing those resources. If this instance of correct behavior is not the intended behavior, then the installer is left to install the external resources and then re-configure, make, and install Babel. This chapter is intended to provide help and reassurance that Babel is indeed configured and installed correctly.

Contents

2.1 Simple Installation

These instructions assume you have a "tarball" (e.g. *.tar.gz file). We have volunteers who put together and manage RedHat RPMs and Debian *.deb distributions of Babel. If you have one of these distros, read their documentation first as it may have details that supersede our own.

A typical build is a simple sequence of

```
% ./configure
# lots of stuff
...
Fortran77 enabled.
C++ enabled.
Java enabled.
Python enabled.
Fortran90 enabled.
```

```
% make
# lots more stuff
...
% make install
# not so much stuff
...
```

There are many circumstances where the configuration step will properly terminate with an error, but if the configuration works, the build and installation shouldn't terminate abnormally. If you have problems or note bugs during configuration, installation or later Babel usage, please send an email to babel-bugs@cca-forum.org including the version of babel you are working with, if possible the output from **babel-config --version-full**, and the exact output that indicates the presence of a bug. **babel-config** is Bourne shell script that the configure creates in the bin directory. If your current directory is the top directory of the Babel distribution, normally you can invoke **babel-config** as follows:

```
% bin/babel-config --version-full
```

2.1.1 Configure

There are two main choices to be made at configure time: "Where does the software get built?" and "Where does the software get installed?". The mechanisms for effecting these choices are quite different.

If you want to build software in a separate directory from where the tarball was untarred, this is called a "VPATH build". VPATH builds are useful if you want to build Babel multiple times with various compilers, flags, or you have a shared filesystem across multiple platforms. It separates the code you generate from things that you were given. The downside is that its more complex to remember where to edit what since original sources will be in the source directory tree and the generated sources and compiled assets will be in the build directory tree.

If you run configure in the directory it appears, (i.e. you typed **./configure**) you are performing an "non-VPATH build". To do a VPATH build, simply cd to the directory you want to be the build directory root, then launch configure from there. The following sequence demonstrates a vpath build

```
% tar zxvf babel-x.x.x.tar.gz
% mkdir babel-linux-build
% cd babel-linux-build
% ../babel-x.x.x/configure
```

Note that the directory where you build Babel should be different from the directory where you install Babel. The default install directory is /usr/local, but can be set to any directory that you have read/write access to. To change the install directory, run configure with the **--prefix** option. Since many people do not have root access on their machine (or prefer to install in a local directory when dealing with unfamiliar software), this option is probably the second most heavily used option for configure (first being **--help**, which is a good one to try also.)

At the time of this writing (0.9.3), there are two configure scripts in Babel, about 40K lines of shell script each. These configure scripts will then propagate the information they acquire to Makefiles by perform approximately 190 sed substitutions (per Makefile), to the source code by setting approximately 170 preprocessor macros in babel_config.h, and various bits of shell script in the build that do not get propagated to the install directory. The configure script does not modify any source code in Babel's runtime system or code generator. This means that source code generated by a different Babel installation is usable as long as it gets compiled against the local babel_config.h and linked with the local Babel runtime libraries.

2.1.2 Make

The makefiles are generated by the configure script from Makefile.in templates. The configure script is generated by a tool called autoconf. The Makefile.in's are generated from Makefile.am files by a separate, related tool called automake. We also use a tool called libtool to help with libraries. Libtool is written in shell, automake in perl, and autoconf in m4.

After a successful configuration step, if your build fails it is most likely that there is a bug in Babel, autoconf, libtool, or a library of m4 macros from any of the above. It is less likely to be an issue with automake, but possible. Perl and m4 themselves are no longer involved in the process after the configure script is produced, so while there may be a nascent bug in the files they generated, it is unlikely.

2.1.3 Make Check (Optional)

This is an exhaustive check that can take hours to complete on an average workstation. The number of actual tests run depends on the number of languages that are enabled. In general a driver and an implementation of each test is generated in each enabled language. Then each combination of driver and implementation are run (both statically linked libraries and dynamically loaded libraries, as appropriate) and tested. A test script can actually launch multiple tests, and tests can have multiple parts. At the time of this writing (babel-0.9.3) there are over 13,000 parts tested when all languages are enabled.

2.1.4 Make Install

This transfers built software to the final installation directory. Examples and tests are not installed, nor are Makefiles or dozens of other types of files. Make install also builds javadoc documentation for Babel's code generator. Since some libraries are built with install paths in mind, libtool uses a lot of scripts to make things work in their build directory with binaries actually hidden in .lib subdirectories. Make install strips this extra scaffolding away as well.

2.1.5 Make Installcheck (Optional)

This is the same test suite as with make check. The only difference is that it is run against the code in the install directories, not the build directories.

2.2 External Software Requirements

Babel builds on a lot of available software; some optional, some required. Some we ship in our tarball, some we require users to install separately.

2.2.1 Required & Included

- **Java GetOpt:** This is a Java rewrite of GNU GetOpt available at http://www.urbanophile.com/arenn/hacking/download.html. The Babel code generator uses this to parse command line arguments. The JAR file, download information, and licensing details are in the lib/ subdirectory of the Babel distribution.
- **Xerces-J:** Xerces-J is a Java implementation of SAX and DOM XML parsers available from the Apache Software Foundation at http://www.apache.org. The Babel code generator uses this for XML I/O. The JAR file, download information, and licensing details are in the lib/ subdirectory of the Babel distribution.

2.2.2 Required but Separate

- **Unix shell & bintools:** On early 64bit Linux boxes, we found it necessary to rebuild even these basic tools with all 64bit options enabled. Apparently they were originally installed with less attention to detail than necessary. Bintools includes things like cp and mv.
- **C/C++ compiler:** The Babel runtime library and much of the code generated by the Babel code generator will be ANSI C. So that must be available. The C++ compiler should be optional, but at the time of this writing the configure and makefiles didn't reliably support disabling C++.
- **Java:** The Babel code generator is implemented in Java. One can disable the support for Java language bindings, but a working Java would still be needed for just about everything else. We generally stick with Sun's java developer kits (available at http://java.sun.com). Others have run Babel with Kaffe and GJC.

- **libxml2:** This is the Gnome C library for parsing XML files (see http://xmlsoft.org). The Babel runtime library needs version 2.4 or above to parse SCL files for dynamic loading.

2.2.3 Recommended

- **Python:** Needed for the python language binding (obviously) and for the testing harness. Since the Linux kernel is often configured with a Python-based tool, its hard to find a Linux without python already installed. Python can be downloaded from http://www.python.org.

  One important gotcha is a special case where non-python applications create Babel objects implemented in python. In this case, the Babel runtime needs to dynamically load the python virtual machine (libpython.so). Unfortunately, python does not always build a dynamically loadable version of this library by default. If the Babel configure script cannot find a libpython.so, it will disable server-side Python support.

  At the time of this writing, Python cannot be coerced to build a libpython.so on AIX.

- **Numeric Python (NumPy):** This is a scientific array python extension module. It provides native C arrays (and the ability to manipulate very big arrays) similar to python lists. Babel's python language binding requires this extension module available at http://www.pfdubois.com/numpy.

- **Python Meta Widgets (Pmw):** This is a library of GUI widgets built on top of Python's native tcl/tk interface (tkinter). Its available on SourceForge http://pmw.sourceforge.net Pmw is only needed by the GUI in the babel-life supercomputing demo. This Babel implementation of Conway's Game of Life is a separate tarball found in the contrib/ directory of the Babel distro. There is no test for Pmw in Babel's configuration script.

- **Chasm:** Babel uses the Fortran array descriptor library available in Chasm (see http://chasm-interop.sourceforge.net). Chasm is a language interoperability tool in its own right, but as of version 1.0.1, only the array library is considered complete. Without Chasm, the configuration script will disable Fortran 90 support.

- **pthreads:** Needed for Java language binding.

2.2.4 Optional

These packages are used by Babel maintainers in the course of normal development. You'll need these only if you start rewriting code in Babel's distribution.

- **Automake:** Part of GNU Autotools (see http://www.gnu.org/software/automake). Check the configure.ac file to determine exactly which version we use. The configure script will disable autoconf if it detects the slightest variation from the version we prescribe.

- **Autoconf:** Part of GNU Autotoolssee http://www.gnu.org/software/automake). Check the configure.ac file to determine exactly which version we use. The configure script will disable autoconf if it detects the slightest variation from the version we prescribe.

- **Libtool:** Part of GNU Autotools (see http://www.gnu.org/software/libtool). Note that we often find need to make minor tweeks to ltmain.sh so a fresh download may generate slightly worse results on some platforms.

- **m4:** Contact us for a patched version that we use (we overflow buffers in the distributed version).

- **JavaCC:** This Java Compiler Compiler is what we use to generate the SIDL parser in Babel. If you are interested in experimenting with changing the SIDL grammar, then edit the compiler/gov/llnl/babel/parsers/sidl/sidl.jj file and rebuilt the parser with this tool. Information available at https://javacc.dev.java.net.

- **LaTeX2HTML:** This is used to generate HTML the HTML version of our manuals.

- **perl:** Needed by automake, LaTeX2HTML and other bits and pieces.

Chapter 3

Basic Babel Code Generation

This chapter describes the Babel code generator and its command line options.

Contents

3.1 Babel is a Compiler

Babel is a compiler. It takes symbols and their interfaces as input and generates either code or a given textual representation. These interfaces may be specified in either Scientific Interface Definition Language (SIDL) or Extensible Markup Language (XML). The form the output takes depends upon the options specified on the command line. Refer to the Section 3.2 for details on command line options. More information on the supported bindings can be found in Part II of this document.

3.2 Command Line Options

The entire Babel code generator is written in Java and compiled into a jar file. For convenience, a small script called **`babel`** is provided that *should* set the appropriate environment variables and invoke the Java Virtual Machine on the jar file. To test that the script and jar file are working together properly, simply type **`babel --help`**.

Using Babel

Babel requires exactly one of the following mutually exclusive arguments on the command line unless you use the **`--multi`** option.

- **`--help`** : Print options to stdout.
- **`--version`** : Print version of Babel.
- **`--text=`*`form`*** : Generate text equivalent ("sidl" or "xml") of associated package(s) or generate interface documentation with "html".
- **`--client=`*`lang`*** : Generate client, or proxy, classes to access library.
- **`--server=`*`lang`*** : Generate the server and client classes to implement the library.
- **`--parse-check`** : Check the SIDL file only.

By far, the three most common uses of Babel will be to generate the Client-side proxies, Server-side implementations, and XML associated with the SIDL file. The last option is essentially used internally when the Babel runtime library is being developed.

The **--multi** option lets you generate multiple targets for a given set of files in a single run. Put it first on the command line, each **--client** or **--server** can have a different set of settings.

Additionally, there are a few supplemental arguments that complete the picture.

- **--assertion-level=*level*** : Generate SIDL-specified assertion checking code. The assertion checking code is part of our Babel research. A *level* of 0 means no assertion checking, 1 means basic support and 2 is advanced.
- **--output-directory=*dir*** : Specifies the root directory associated with the generated files. The default setting is the current working directory.
- **--generate-hooks** : Generate additional methods in the implementation files that allow developers to put additional code to be called before and after the actual method call. These hooks are useful for automatically instrumenting code.
- **--generate-subdirs** : Generates files in a directory tree matching the packaging scope of the SIDL file. This is on by default for languages that have this requirement, such as Java and Python, but off by default for languages that have no such requirement. Hence, code generation for only the latter languages (e.g. C, C++, F77, F90) is effected by this option.
- **--generate-subdirs-off** : Turn the **--generate-subdirs** feature off. This is useful with **--multi** option.
- **--short-file-name** : When the --generate-subdirs and --short-file-names options are used simultaneously, the generated file names will not include package names, just the class or interface symbol. Thus, either long or short names must be used in all clients or servers that have interdependencies; mixing short and long names will result in compile and/or runtime errors.
- **--repository-path=*path*** : Specifies a semicolon separated list of directories, or URLs[1] to search for XML Type descriptions. The need for these XML types is to resolve references in the SIDL file. This option can be used multiple times on the same command line. If appropriate, the Babel script adds the default repository path to the command line before dispatching to the Java Virtual Machine.
- **--no-default-repository** : Prohibits the use of the default repository in resolving symbols.
- **--suppress-ior** : Refrain from generating IOR source and header files.
- **--suppress-timestamp** : Suppresses the insertion of meta-information that could result in generated files that would otherwise not differ from prior executions on the same, unchanged input file. Typically Babel inserts meta-information such as creation time into files it generates. Although this information is useful, it does result in the creation of excessive changes when using version control systems.
- **--exclude=*regex*** : This options can be used multiple times. Each time you add a regular expression that will be used to exclude symbols from code generation. No code or XML will be generated for any symbol matching the user provided regular expression. This command line option requires version 1.4.0 or later of the Java runtime environment.
- **--comment-local-only** : This option reduces the amount of comments in stub C header files. It will only include the doc comments for locally defined method. It will not include doc comments for inherited methods.
- **--hide-glue** : This option causes all non-impl files to be generated in a `glue/` subdirectory. This reduces the "clutter" in the current directory.
- **--hide-glue-off** : Turn off the **--hide-glue** setting.

[1] URLs have colons in them, so this path has to be semi-colon separated, even though UNIX paths are traditionally colon separated.

- `--language-subdir` : This options causes all generated files to be stored in a language-dependent subdirectory; if the `--generate-subdirs` option is also used, the language directory will be at the bottom of the hierarchy.
- `--language-subdir-off` : Turn the `--language-subdir` setting off.
- `--make-prefix=`***prefix*** : The string *prefix* is prepended to the name of `babel.make` and the symbols defined inside `babel.make` to allow Babel to be run multiple times in a single directory without overwriting files.
- `--exclude-external` : This option causes code to be generated only for the symbols specified on the command line. If you list a SIDL file on the command line, all the symbols it contains will be included. No code is generated for symbols on which the users symbols depend.
- `--cxx-ior-exception` : Earlier versions of the Babel C++ bindings checked the IOR pointer in a given stub before making any calls on it. If the IOR was null, a NullIORException was thrown. It was later found that in certain cases these checks were taking an inordinent amount of time, and since C++ does not normally check pointers before dereferencing them, it was decided that this feature was out of line with the spirit of C++. However, since some code had already been written that used this feature, we could not completely eliminate the checks. Therefore, this command line option was added. Calling babel with it will generate C++ stubs with the checks in them. This option has no effect on other languages.
- `--vpath=`***dir*** : This option sets the root directory Babel searches first when trying to load implementation files to preserve splicer block contents in the hand edited implementation files. If you are generating server-side C for a concrete class `x.y.z` and you used `--vpath=/tmp`, Babel would try to read splicer blocks from `/tmp/x_y_z_Impl.h` and `/tmp/x_y_z_Impl.c`. If it does not find either file in `/tmp`, it also checks the current directory. If you are using `--generate-subdirs` with `--vpath`, the vpath directory is the root of the tree, so for the example, Babel would search for `/tmp/x/y/z_Impl.h` and `/tmp/x/y/z_Impl.c`. When appropriate, Babel inserts `#line` directives to refer debuggers to the original file. As its name suggests, this option is useful when making vpath builds using make. Some people also use it to avoid spurious changes to the files managed by their revision control system.

Long and Short Forms

So far, we've shown described the long forms of command line arguments, starting with two hyphens "--". There are also short forms for many of the more frequently used commands. See Table 3.1 for details.

Examples

To create a new XML version of a SIDL file, use the following command:

```
% babel -E -tXML -omydepot mystuff.sidl
```

To exclude code generation for types whose name begins with "MPI.", use the following command:

```
% babel -sUC++ --exclude='^MPI\.'  mystuff.sidl
```

Note, at this time, there are two C++ bindings in Babel. The current and future binding is called `UC++` or `UCxx`, and the old, deprecated binding is called `DC++` or `DCxx`. This manual will focus on the `UC++` binding.

Now suppose a developer wants to implement a library in C++ that corresponds to these types in the SIDL file.

```
% babel -E -sUC++ mystuff.sidl
```

Alternatively, the developer could also create C++ implementation files based on the XML repository. In this case, a list of symbols to be implemented would need to be specified. Assuming that all of the types are in a package called "mystuff", the following command can be issued:

Table 3.1: Command Line Arguments.

SHORT FORM	LONG FORM	NOTES
-h	−−help	Print options to stdout.
-v	−−version	Print version of Babel.
-a	−−assertion-level=*level*	Generate assertion checks.
-t*form*	−−text=*form*	Generate text.
-c*lang*	−−client=*lang*	Generate client classes.
-s*lang*	−−server=*lang*	Generate server and client classes.
-p	−−parse-check	Only check parsing of the SIDL file.
-i	−−generate-hooks	Generate pre-/post-method hooks.
	−−generate-sidl-stdlib	Regenerate the Babel runtime library.
-o*dir*	−−output-directory=*dir*	Root directory to contain generated files.
-g	−−generate-subdirs	Generate sources in directory tree matching SIDL packaging.
	−−generate-subdirs-off	Turn off generate-subdirs.
-m	−−make-prefix=*prefix*	Prepend *prefix* to names for babel.make.
	−−multi	Generate multiple targets in a single run.
-R*path*	−−output-directory=*path*	Use specified XML repository(ies) to resolve symbols.
-e*regex*	−−exclude=*regex*	Do not generate output for matching symbol(s).
	−−no-default-repository	Do not use the default repository to resolve symbols.
	−−suppress-ior	Don't generate IOR files.
	−−suppress-timestamp	Suppress time-related metadata generation.
	−−comment-local-only	Reduce doc comments in C stub header.
-E	−−exclude-external	Do not generate code for dependencies.
-u	−−hide-glue	Put glue code in a subdirectory.
	−−hide-glue-off	Turn off hide-glue.
-l	−−language-subdir	Put code in a language dependent directory.
	−−language-subdir-off	Turn off language-subdir.
-x	−−cxx-ior-exception	Include Null IOR checks in C++ Stubs.
-V	−−vpath	Set the impl (splicer block) root directory.

```
% babel -E -sUC++ -Rmydepot mystuff
```

Now suppose a second developer wants to extend this software. A second SIDL file is created then the implementation files in FORTRAN 90 are generated with the following command:

```
% babel -E -sf90 -Rmydepot newstuff.sidl
```

A user now can download both SIDL files and create their Python bindings to use both libraries with the following command:

```
% babel -cPython -Rhttp://localhost/mystuff/mydepot;
http://www.otherhost.com/newstuff mystuff newstuff
```

Finally, to generate SIDL files for each package based on the XML stored in the repository, the following command is used:

```
% babel -tSIDL -Rhttp://localhost/mystuff/mydepot;
http://www.otherhost.com/newstuff mystuff newstuff
```

Chapter 4

Hello World Tutorial

For the things we have to learn before we can do them, we learn by doing them.
— Aristotle (384 BCE – 322 BCE)

Contents

4.1 Introduction

This tutorial guides you through the process of writing the classic "Hello World!" example using the Babel tools. In the process, you will learn that the most vexing problem is getting the compiler and linker flags set up properly. Closely followed by the hassles of encoding this information in portable Makefiles.

This section offers a spectrum of possible solutions, from the minimalist but not portable, to the very robust and portable but not trivial, and a few options in between. The following sections start from the simplest setup, and work up in terms of complexity and features.

Assuming Babel is built, installed and your PATH environment variable has been set, we need to set up a few directories for this exercise. One can verify it was built and installed properly by going to the directory where it was built and typing `make installcheck`. (Warning: It can take a few hours on a good workstation for Babel's exhaustive tests to complete.) To verify Babel is in your path, simply try running it with the `--version` or `--help` option. Now pick a starting directory and issue the following commands to create some directories

```
% mkdir -p hello/minimal/libCxx
% mkdir -p hello/minimal/libF90
% cd hello
```

Now we write a SIDL file to describe the calling interface. It will be used by the Babel tools to generate glue code that hooks together different programming languages. A complete description of SIDL can be found in Chapter 5. We will use the same SIDL file for several different coding exercises and Makefile setups, so the SIDL file need not be large or complex for our purposes.

For this particular application, we will write a SIDL file that contains a class World in a package Hello. Method getMsg() in class World returns a string containing the traditional computer greeting. Using your favorite text editor, create a file called hello.sidl in the hello/ directory containing the following:

SIDL

```
1 package Hello version 1.0 {
2   class World {
3     string getMsg();
4   }
5 }
```

The package statement provides a scope (or namespace) for class World, which contains only one method, getMsg(). The version clause of the statement identifies this as version 1.0 of the Hello package.

4.2 Minimal Makefiles: static linked and unportable

This section will show a C driver that is used to call both C++ and Fortran 90 implementations using static linked libraries. The makefiles are hand-coded specificly for GCC 4.0 compilers (or better) on Linux. If your setup is different, you can try to adjust the example to suit your situation, or skip to section 4.3 which involves more setup, but is also more portable.

4.2.1 Writing the C++ Implementation

We will write the C++ implementation in the minimal/libCxx/ subdirectory of hello/. The first step is to run the Babel code generator on the SIDL file and have it generate the appropriate code. The simplified command to generate the Babel library code (assuming Babel is in your PATH) is [1]:

```
% cd minimal/libCxx
% babel -sC++ ../../hello.sidl
```

In this Babel command, the "`-sC++`" flag, or its long form "`--server=C++`", indicates that we wish to generate C++ bindings for an implementation[2]. This command will generate a large number of C and C++ header and source files. It is often surprising to newcomers just how much code is generated by Babel. Rest assured, each file has a purpose and there is a lot of important things being done as efficiently as possible under the hood.

Files are named after the fully-qualified class-name. For instance, a package *Hello* and class *World* would have a fully qualified name (in SIDL) as *Hello.World*. This corresponds to file names beginning with `Hello_World`[3]. For each class, there will be files with `_IOR`, `_Skel`, or `_Impl` appended after the fully qualified name. Stubs often go without the `_Stub` suffix. *IOR files* are always in ANSI C (source and headers), containing Babel's Intermediate Object Representation. *Impl files* contain the actual implementation, and can be in any language that Babel supports, in this case, they're C++ files. Impl files are the only files that a developer need look at or touch after generating code from the SIDL source. *Skel files* perform translations between the IORs and the Impls. In some cases (like Fortran) the Skels are split into a few files: some in C, some in the Impl language. In the case of C++, the Skels are pure C++ code wrapped in `extern "C" {}` declarations. If the file is neither an IOR, Skel, nor Impl, then it is likely a *Stub*. Stubs are the proxy classes of Babel, performing translations between the caller language and the IOR. Finally, the file babel.make is a Makefile fragment that will simplify writing the Makefile necessary to compile the library. You may ignore the babel.make file if you wish. There are also babel.make.depends and babel.make.package files. These were added by external contributors, and we will ignore them in thid document.

The only files that should be modified by the developer (that's you since you're implementing Hello World) are the "Impls", which are in this case files ending with `_Impl.hxx` or `_Impl.cxx` Babel generates these implementation files as a starting point for developers. These files will contain the implementation of the Hello library. Every implementation file contains many pairs of comment "splicer" lines such lines 4 and 6 in the following sample:

[1] For information on additional command line options, refer to Section 3.2.

[2] You can also try the "`--help`" flag to list all of the Babel command-line options.

[3] Note: dots are converted to underscores for file naming.

```
::std::string
Hello::World_impl::getMsg_impl ()
{
  // DO-NOT-DELETE splicer.begin(Hello.World.getMsg)
  // Insert-Code-Here {Hello.World.getMsg} (getMsg method)
  // DO-NOT-DELETE splicer.end(Hello.World.getMsg)
}
```
C++

Any modifications between these splicer lines will be saved after subsequent invocations of the Babel tool. Any changes outside the splicer lines will be lost. This splicer feature was developed to make it easy to do incremental development using Babel. By keeping your edits within the splicer blocks, you can add new methods to the hello.sidl file and rerun Babel without the loss of your previous method implementations. You shouldn't ever need to edit the file outside the splicer blocks.

For our hello application, the implementation is trivial. Add the following return statement between the splicer lines in the `lib/Hello_World_Impl.cxx` file:

```
::std::string
Hello::World_impl::getMsg_impl ()
{
  // DO-NOT-DELETE splicer.begin(Hello.World.getMsg)
  return "Hello from C++!";
  // DO-NOT-DELETE splicer.end(Hello.World.getMsg)
}
```
C++

To keep the Makefile minimal, the GNU gcc and g++ compilers are hard coded into this example. Put the following Makefile in the minimal/libCxx subdirectory to compile the files and create a linked library named `libhello.a`:

NOTE:

Static libraries will limit us to compiled languages (e.g. C/C++ & Fortran). To use compiled implementations in more dynamic languages like Java and Python, they must be built as dynamic loaded libraries (a.k.a. run-time linking in some circles). Building these kinds of libraries is harder and often misunderstood, so we will postpone them until later.

```
# A minimal makefile for a C++ Babel Impl.
# Assumes babel-config is in the current path.
# Assumes babel -sC++ ../../hello.sidl has already run.

include babel.make

OBJS = $(IMPLSRCS:.cxx=.o) $(IORSRCS:.c=.o) \
        $(SKELSRCS:.cxx=.o) $(STUBSRCS:.cxx=.o)

all: libhello.a

.SUFFIXES:
.SUFFIXES: .cxx .c .o
.cxx.o:
        g++ `babel-config --includes-c++` -c $<
.c.o:
        gcc `babel-config --includes-c++` -c $<

libhello.a: $(OBJS)
        ar cru $@ $(OBJS)
```
make

```
21          ranlib $@
22
23  .PHONY: clean new
24
25  clean:
26          $(RM) *.o babel.make.depends babel.make.package *~
27
28  new: clean
29          $(RM) $(IORHDRS) $(IORSRCS) $(SKELSRCS) $(STUBHDRS) \
30          $(STUBSRCS) libhello.a
```

The details of this makefile deserve careful explanation.

line 5: `babel.make` is a file that Babel generates when it generates code. It defines some standard names so our makefiles don't have to know every SIDL type declared in the file.

lines 7–8: Here you see some of the variables defined in `babel.make` and some fancy suffix substitution so that the makefile variable `$(OBJS)` is a list of all `.o` files.

line 10: The first build target in a Makefile is also the default target. Common convention is to make this target's name "all."

line 12–17: Here we start to override make's default suffix rules for converting C and C++ sourcecode into `.o` files.

line 12: This is a often misunderstood detail. To override the default suffix rules, one must give the list of new suffices to consider. But before that, one must wipe out the old suffices as we do here.

lines 15 & 17: Note that we use a script called `babel-config` to hand us some information. This tool is very useful for getting the same information that Babel's configure script was given as Babel configuring itself. Increasingly, it is being used to also get information about tools and flags used in Babel's makefiles, and its utility for getting at this level of information is limited... mostly because we rely on a multilayered stack of tools and some are less forthcoming than others.

lines 19–21: This is the rule to combine the `.o` files into a static library.

line 23: Technically, any phony targets like "clean" and "new" are supposed to be listed in this variable. Many makefile implementations will work well if you skip this line, but.

lines 25–29: How you go about various levels of cleanliness in your makefiles largely depends on matters of taste. The important point here is what is *not* removed. One thing not cleaned up is `babel.make` it is generated by Babel, but the makefile won't work without the file present because of the include in line 5. Another important thing to *not* remove is the Impl files, since they have hand-edited regions in the generated file.

With the makefile in place we can simply go to that directory and build everything by typing make.

4.2.2 Writing the Fortran 90 Implementation

Before writing the client, let's generate a Fortran implementation as well. It is highly instructive to see how the makefiles differ between the different language bindings. From within the `minimal/libCxx` directory we do.

```
% cd ../libF90
% babel -sF90 ../../hello.sidl
```

This time there's even more files generated (Fortran 90 bindings are harder after all), and we need to add our implementation to the `Hello_World_Impl.F90` file. The modified code will look like this.

Fortran 90

```
1  !
2  ! Method:  getMsg[]
3  !
4
5  recursive subroutine Hello_World_getMsg_mi(self, retval, exception)
6    use sidl
7    use sidl_BaseInterface
8    use sidl_RuntimeException
9    use Hello_World
10   use Hello_World_impl
11   ! DO-NOT-DELETE splicer.begin(Hello.World.getMsg.use)
12   ! Insert-Code-Here {Hello.World.getMsg.use} (use statements)
13   ! DO-NOT-DELETE splicer.end(Hello.World.getMsg.use)
14   implicit none
15   type(Hello_World_t) :: self ! in
16   character (len=*) :: retval ! out
17   type(sidl_BaseInterface_t) :: exception ! out
18
19 ! DO-NOT-DELETE splicer.begin(Hello.World.getMsg)
20 retval='Hello from Fortran 90!'
21 ! DO-NOT-DELETE splicer.end(Hello.World.getMsg)
22 end subroutine Hello_World_getMsg_mi
```

Note that the C function appears as a subroutine in Fortran. What was the return value appears here as the argument `retval` (line 5). For Fortran 90 there are also two splicer blocks per subroutine, one for use statements (lines 11–13) and another for the actual implementation (lines 19–21). This is where we put our implementation by setting `retval` to the string we want.

There are important differences in this Makefile from the C++ implementation, so we reproduce it in its entirety here.

make

```
1  # A minimal makefile for Fortran 90 Babel Impl
2  # Assumes babel-config is in the current path
3  # Assumes babel -sF90 ../../hello.sidl is already run
4
5  include babel.make
6
7  OBJS = $(IORSRCS:.c=.o) $(TYPEMODULESRCS:.F90=.o) \
8         $(SKELSRCS:.c=.o) $(STUBMODULESRCS:.F90=.o) $(STUBSRCS:.c=.o) \
9         $(IMPLMODULESRCS:.F90=.o) $(IMPLSRCS:.F90=.o)
10
11 all: libhello.a
12
13 .SUFFIXES:
14 .SUFFIXES: .F90 .c .o
15 .F90.o:
16         gcc -E -traditional -P -o $*.tmp -x c  \
17                 `babel-config --includes-f90` $<
18         sed -e 's/^#pragma.*$$//' < $*.tmp > $*.f90
19         gfortran -c -o $@ `babel-config --includes-f90-mod` $*.f90
20         rm -f $*.f90 $*.tmp
21
22 .c.o:
23         gcc `babel-config --includes` `babel-config --includes-f90` -c $<
24
25 libhello.a: $(OBJS)
```

```
26          ar cru $@ $(OBJS)
27          ranlib $@
28
29  .PHONY:  clean new
30
31  clean:
32          $(RM) *.o *.mod *~ babel.make.depends babel.make.package
33
34  new: clean
35          $(RM) $(IORHDRS) $(IORSRCS) $(TYPEMODULESRCS) $(SKELSRCS) \
36          $(STUBMODULESRCS) $(STUBSRCS) $(STUBHDRS) libhello.a
```

line 5: Again Babel will generate a `babel.make` file, but we will see that its contents are different.

lines 7–9: Are building a `$(OBJS)` variable like before, but this time we see suffix substitutions for more kinds of files. We caution the reader that Fortran 90's use of MOD files[4] makes the ordering of these items very important. (Not Babel's fault, blame the Fortran 90 language designers.) C/C++ has no such constraint on the order that individual units of compilation are performed. As long as Fortran 90 programmers stick with the ordering shown in these lines, they should not encounter compiler complaints about dependent MOD files not found.

lines 16–20: This bit of code admittedly looks very strange, but the explanation is simple. We preprocess our Fortran 90 source to workaround the 31 character limit specified in the language. Check out Chapter 9 for more details about this issue.

lines 17 & 19: Note that we use `babel-config` to generate the proper flag for the preprocessor to find the Babel Fortran headers, and the compiler to find the Babel MOD files.

line 23: When compiling C source in this directory, we use `babel-config` to generate include directives for both the default (C) and Fortran include directories.

lines 25–end: The rest of the makefile is almost identical. Note that we do *not* clean up the `Hello_World_Mod.F90` or the CodeHello_World_Impl.F90 files.

Again, we simply type make, and should end up with another `libhello.a` file.

4.2.3 Writing the C Client

Now, finally, we are ready to write a client. For this exercise, we wrote our driver in C and built two executables; each one linking in one of the two implementation libraries. We will put our driver in the `minimal/` directory (which happens to be the parent directory of where the C++ and Fortran 90 implementations are, though this detail is only relevant to makefile construction). From our Fortran 90 subdirectory, we go up one and generate the client-side C bindings.

```
% cd ..
% babel -cC ../hello.sidl
```

The "`-cC`" flag, or its equivalent long-form "`--client=C`", tells the Babel code generator to create only the C stub calling code, not the entire library implementation.

There are a few details worth noting here. The C bindings generate function names by combining packages, classes, and method names with underscores (e.g. `Hello_World_getMsg()`. Whenever you see double underscores in Babel generated symbols, they indicate something built-in to (and sometimes specific to) the language binding. The `_create()` method is built-in to every instantiatable class defined in SIDL, triggering the creation of Babel internal data structures as well as the constructor of the actual object implementation.

The code listing below shows a well crafted driver with full error checking.

[4] A kind of precompiled header.

ANSI C

```c
#include <stdio.h>
#include "Hello_World.h"
#include "sidl_BaseInterface.h"
#include "sidl_Exception.h"


int main() {
  Hello_World h;
  sidl_BaseInterface ex;
  char * msg;

  /* create instance of Hello World */
  h = Hello_World__create(&ex); SIDL_CHECK(ex);
  if ( h == NULL ) {
    fprintf(stderr,"%s:%d Failed to create an instance of Hello_World!\n",
            __FILE__,__LINE__);
    return 2;
  }

  /* get the message from the object */
  msg = Hello_World_getMsg(h, &ex); SIDL_CHECK(ex);
  if ( msg == NULL ) {
    fprintf(stderr, "%s:%d Hello_World_getMsg() returned a NULL instead "
                    "of a string!\n",__FILE__,__LINE__);
    return 3;
  }

  /* done with object so we can release it */
  Hello_World_deleteRef(h,&ex); SIDL_CHECK(ex);

  /* print the string */
  printf("%s\n",msg);

  /* release the string */
  sidl_String_free(msg);

  return 0;

 EXIT: /* this is error handling code for any exceptions that were thrown */
  {
    fprintf(stderr,"%s:%d: Error, exception caught\n",__FILE__,__LINE__);
    sidl_BaseInterface ignore = NULL;
    sidl_BaseException be = sidl_BaseException__cast(ex,&ignore);

    msg = sidl_BaseException_getNote(be, &ignore);
    fprintf(stderr,"%s\n",msg);
    sidl_String_free(msg);

    msg = sidl_BaseException_getTrace(be, &ignore);
    fprintf(stderr,"%s\n",msg);
    sidl_String_free(msg);

    sidl_BaseException_deleteRef(be, &ignore);
    SIDL_CLEAR(ex);
```

```
55      return 1;
56    }
57  }
```

As with other examples, we will go through this one line by line. It is important to note that no where in this file is any indication of what language the Babel object is implemented in. When you see the makefile, we will show that this code can be linked against multiple implementations in different languages.

line 2: This line includes the C stub for the *Hello.World* type.

line 3: We also include the C stub for the *sidl.BaseInterface* type which all classes and interfaces ultimately inherit from. In C, we often use this type to hold the exception argument.

line 4: There is no *sidl.Exception* type. This header actually introduces some useful macros for dealing with exception handling in C.

line 13: This is where the object is `_create()`'ed. Note that the creation may fail, so we use the `SIDL_CHECK` macro introduced from `sidl_Exception.h` to test the exception and goto `EXIT` (line 39), if necessary.

line 21: With a live reference to the object, we now try to get the message out of it. Note that we check if the exception is thrown *and* if the string is `NULL`.

line 29: Once we have the message, we can dispose of our reference to the object.

line 32: Print the message

line 35: Free the string. Return values have the same semantics as out parameters which is the caller always recieves a reference count and is obligied to dispose of it when done.

line 37: Normal termination.

lines 39–56: This is exception handling code. Its hard to imagine so many possibilities for failure in our little example, but it is useful to see how exception classes can be cast to appropriate types (line 43), and be queried for both original error message and the trace of the call stack from which it was thrown.

line 42: Note that Babel generated methods always throw exceptions, but in exception handling code, we often ignore them. Do not call `SIDL_CHECK` after the `EXIT` as this can easily result in an infinite loop.

Now we need to write a makefile that builds the code in this directory and links it with the C++ or Fortran 90 implementations in the two subdirectories.

make

```
1  # A minimal makefile for Fortran 90 Babel Impl
2  # Assumes babel-config is in the current path
3  # Assumes babel -cC ../hello.sidl is already run
4
5  include babel.make
6
7  LIBDIR=`babel-config --libdir`
8  LIBS=-lsidlstub_cxx -lsidlstub_f90 -lsidl
9  OBJS = main.o ${STUBSRCS:.c=.o}
10
11 all: runC2Cxx runC2F90
12
13 .SUFFIXES:
14 .SUFFIXES: .c .o
15
16 .c.o:
17         gcc `babel-config --includes` -c $<
18
```

```
19 runC2Cxx: ${OBJS} libCxx/libhello.a
20         g++ -o runC2Cxx $(OBJS) libCxx/libhello.a \
21                 -L$(LIBDIR) `babel-config --libs-f90` $(LIBS)
22
23 runC2F90: ${OBJS} libF90/libhello.a
24         g++ -o runC2F90 $(OBJS) libF90/libhello.a \
25                 -L$(LIBDIR) `babel-config --libs-f90` $(LIBS)
26
27 .PHONY:  clean new
28
29 clean:
30         ${RM} *.o *~ babel.make.depends babel.make.package
31
32 new: clean
33         $(RM) $(IORHDRS) $(STUBSRCS) $(STUBHDRS) runC2Cxx runC2F90
```

line 5: Again we include the Babel-generated makefile fragment. Again we see that its contents depend on the language being generated.

line 6: We use `babel-config` to find out what directory Babel was configured to put its libraries in.

line 7: We create a makefile macro to link in some of Babel's default libraries. `libsidl.{a,so}` is *the* Babel runtime library. All Babel code ultimately depends on this because it includes the implementation of *`sidl.BaseClass`*. `libsidlstub_cxx` is a library of C++ stubs and `libsidlstub_f90` is a library of F90 stubs to `libsidl`.

line 8: Since client side only includes Stubs and IOR.h, there is much less to compile.

line 11: Default for this makefile is to create two executables.

lines 19–25: Note that we are using the C++ compiler as our linker. This is generally the safest bet because if there's any C++ template code in your combined application, chances are the C++ compiler will want to do template instantiation at link time. `babel-config` can provide all the library flags that the Fortran 90 compiler silently passes on to the linker.

At last, we can make the two executables and run them.

```
% make all
% ./runC2Cxx
Hello from C++!
% ./runC2F90
Hello from Fortran 90!
```

4.3 Portable Makefiles for Static Linkage: using `babel-config`

In this section we look back at the makefile so far and realize there's an awful lot of compiler and platform specfic details in our minimalist makefiles. While simple is good, nonportable is often bad enough to tread into difficult waters.

Since Babelized software must be built the same way that Babel itself was configured, it seems reasonable to lean on `babel-config` quite heavily. By now you are probably wondering how many secrets `babel-config` holds and can provide on request. The simplest way to find out is to ask it. (Though you may get a slightly different result than what is shown here depending on the version of Babel.)

```
% babel-config --dump-vars | wc -l
120
```

4.4 Portable Makefiles and Runtime Linking: adding babel-libtool

4.5 Small and Portable Makefiles: using babel-cc

4.6 Final Remarks

Congratulations! You are now ready to develop a parallel scalable linear solver package.

The preceding process may seem to be the most complicated way to write the world's simplest program but, of course, the same process will also work for significantly more complex applications. "Hello World" is small enough to experiment with in the language of your choice. Parallel, multithreaded, scientific simulation codes are another matter entirely.

Chapter 5

SIDL Basics

Contents

5.1 Introduction

This chapter describes the basics of the Scientific Interface Definition Language (SIDL). The goal is to provide sufficient information to enable most library and component developers to begin using SIDL to wrap their software. It begins with an overview of SIDL files followed by an introduction to the fundamental data types. More complex topics such as the object arrays, exceptions, objects, and the XML repository are then addressed.

5.2 SIDL Files

SIDL files are human-readable, language- and platform- independent interface specifications for objects and their methods. SIDL allows you to specify classes, interfaces, and the methods therein. All methods defined in SIDL are public, since the developer is writing them as part of an interface description. Any data you wish a SIDL object to hold is not declared in the SIDL file, and is private. Data should be placed in the implementation skeleton files, and cannot be publicly exported.

Babel reads the SIDL files to generate the appropriate programming language bindings. These bindings, in the form of stub, intermediate object representation (IOR), and implementation skeleton sources, provide the basis for language interoperable software using Babel. In addition, SIDL files are used to populate the XML symbol repository that can serve as an alternate source of interface specifications during the generation of programming language bindings.

Basic Structure

The basic structure of a SIDL file is illustrated below.

SIDL

```
package <identifier> [version <version>]
{
  interface <identifier> [ <inheritance> ]
```

```
  {
    [<type>] <identifier> ( [<parameters>] ) [throws <exception>];
      .
      .
      .

    [<type>] <identifier> ( [<parameters>] ) [throws <exception>];
  }

  class <identifier> [ <inheritance> ]
  {
    [<type>] <identifier> (<parameters>) [throws <exception>];
      .
      .
      .

    [<type>] <identifier> ( [<parameters>] ) [throws <exception>];
  }

  package <identifier> [version <version>]
  {
      .
      .
      .
  }
}
```

The main elements are *packages*, *interfaces*, *classes*, *methods*, and *types*. For a more detailed description, refer to Appendix B.

Packages provide a mechanism for specifying name space hierarchies. That is, it enables grouping sets of interface and/or class descriptions as well as nested packages. Identified by the *package* keyword, packages have a *scoped* name that consists of one or more identifiers, or name strings, separated by a period (“.”). A package can contain multiple interfaces, classes and nested packages. By default, packages are now re-entrant. In order to make them non-re-entrant, they must be declared as `final`.

Interfaces define a set of methods that a caller can invoke on an object of a class that implements the methods. Multiple inheritance of interfaces is supported, which means an interface or a class can be derived from one or more interfaces.

Classes also define a set of methods that a caller can invoke on an object. A class can extend only one other class but it can implement multiple interfaces. So we have single inheritance of classes and multiple inheritance of interfaces.

Methods define services that are available for invocation by a caller. The signature of the method consists of the return *type*, identifier, arguments, and exceptions. Each parameter has a *type* and a *mode*. The *mode* indicates whether the value of the specified *type* is passed from caller to callee (*in*), from callee to caller (*out*), or both (*inout*). All methods are implicitly capable of throwing a `sidl.RuntimeException` exception. A `sidl.RuntimeException` is used to indicate an error in the Babel generated code or potentially a network exception. Each additional exception that a method can *throw* when it detects an error must be listed. These exceptions can be either interfaces or classes so long as they inherit from `sidl.BaseException`. For a default

implementation of the exception interfaces, the exception classes should extend `sidl.SIDLException`. Methods and parameter passing modes are discussed in greater detail in Section 5.6.

Types are used to constrain the the values of parameters, exceptions, and return values associated with methods. SIDL supports basic types such as `int`, `bool`, and `long` as well as strings, complex numbers, classes, and arrays.

Comments and Doc-Comments

SIDL has the same commenting style as C++/Java and even has a special documentation comment (so called *doc-comment*) similar to those used in Javadoc. One can embed comments anywhere in their SIDL file. Documentation comments should immediately precede the class, interface, or method with which they are associated. Babel replicates documentation comments in the files it generates. It does not replicate plain comments.

SIDL

```
/*
 *  1. This is a multi-line comment.
 *
 */

//  2. This comment fits entirely on a single line.

/*  3. This comment can fill less than a line. */

/** 4. This is a documentation comment. */

/**
 *  5. Documentation comments can span
 *     multiple lines without the beginning
 *     space-asterisk-space combinations
 *     getting in the way.
 */
```

Consider the above SIDL file fragment.

1. This comment is a regular multi-line comment that is delimited by a slash-star , star-slash (“/*”, “*/”) pair.
2. This is a single-line comment that starts with a double slash “//” and continues to the end of the line.
3. This comment is the same as # 1 except that it is completely contained on a single line. It can be embedded in the middle of a line anywhere a space naturally occurs.
4. This is a documentation comment. In keeping with Javadoc, Doc++, and other tools, it is delimited by slash-star-star and star-slash (“/**”, “*/”) combinations. Documentation comments are important because their contents are preserved by Babel in the corresponding generated files. Doc-comments must directly precede the interface, class, or method that they document.
5. This is a multi-line variant of a doc-comment. Note that initial asterisks on a line are assumed to be for human readers only and are discarded by Babel when it reads in the text. The multi-line doc-comment is the preferred way of documenting SIDL.

Packages and Versions

SIDL has both a packaging and versioning mechanism built in. Packages are essentially named scopes, serving a similar function as Java packages or C++ namespaces. Versions are decimal separated integer values where it is assumed larger numbers imply more recent versions. All classes and interfaces in that package get that same version number. If subpackages are specified, they can have their own version number assigned. If a package is declared without a version, it can only contain other packages. If a package declares interfaces or classes, a version number for that package is required.

```
package mypkg {

}
```
SIDL

This SIDL file represents the minimum needed for each and every SIDL file. The package statement defines a scope where all classes within the package must reside. Since no version clause is included, the version number defaults to 0.

Packages can be nested. This is shown in the example below. The version numbers assigned to all the types is determined by the package, or subpackage, in which it resides. In the design of the SIDL file, remember that some languages get very long function names from excessively nested packages or excessively long package names.

```
package mypkg version 1.0 {

  package thisIsAReallyLongPackageName {
  }

  package this version 0.6 {
    package is {
      package a {
        package really {
          package deeply version 0.4 {
            package nested {
              package packageName version 0.1 {
              }
            }
          }
        }
      }
    }
  }
}
```
SIDL

In SIDL you can use as much or as little of a type name as necessary to uniquely identify it. If absolute specificity is required, a leading dot can be used to identify the global (top) package.

```
1  package foo {
2    class A {}
3    package foo {
4      class A {
5        foo.A bar();  //which foo.A?
6        .foo.A wuux(); //first foo.A.
7        .foo.foo.A wuxx(); //second foo.A.
8      }
9    }
10 }
```
SIDL

External types can be expressed in one of two ways. The fully scoped external type can be used anywhere in the class description. Alternatively, an `import` statement can be used to put the type in the local package-space. `import` statements can request a specific version of the package, if that version is not found, Babel will print an error. If no version is specified, Babel will take whatever version it is being run on. Babel can not be run on two versions of a given package at the same time, even if you only import or require one of them.

Another way to restrict the package version you use is the `restrict` statement. `restrict` does not import the package, but if you do later import the package or refer to something in that package by it's fully scoped name, Babel will guarantee that the correct version of the package will be used. Also note that all restrict statements must come before the first import statement.

Below is a sample SIDL file, that should help bring all of these concepts together.

```
require pkgC version 2.0; // restrict pkgC to version 2.0, not imported

import pkgA version 1.0; // restrict pkgA version 1.0. Includes class pkgA.A

import pkgB;      // import pkgB regaurdless of version.  Includes class pkgB.B

package mypkg version 2.0 {
  class foo {
    setA( A ); // imported from pkgA, must be pkgA.A-v1.0
    setB( B ); // imported from pkgB, must be pkgB.B, no version restriction
    setC( pkgC.C ); // must be pkgC.C-v2.0
    setD( pkgD.D ); // no version restriction
  }
}
```
SIDL

Re-entrant Packages

By default, SIDL packages are re-entrant. This means that Babel allows sub-packages to be broken into separate files, but you'd still have to run Babel on all the files at the same time. Here's how it works.

First define the outermost package in a file.

```
package mypkg version 2.0 {

}
```
SIDL

Then define a sub-package in a second file.

```
package mypkg.subpkg version 2.0 {

}
```
SIDL

Note that both files begin with the identical version statement. Now as long as you run Babel on both SIDL files at the same time (with the outermost one first on the commandline), all is fine.

This works because the package statement takes a scoped identifier as an argument. As long as Babel knows that a package *mypkg* exists, it can handle a new package called *subpkg*. (This would also work if *subpkg* were a class. Version statements require an identifier for the outermost package. Since packages cannot have dots "." in their names, the only dots in version statements should appear at the numbers, not the package names.

Running the second file without the first will (and should) generate an error since the enclosing package was not declared. Re-entrance should be used judiciously. This feature may be disabled by labeling a given package as *final*.

The From Clause

The from clause is a special SIDL statement that allows an implementor of multiple interfaces to add or rename the extensions of conflicting methods from interfaces. However, only method extensions can be changed, and the methods must have different signatures. For example, one can change the name of conflicting methods from two interfaces:

```
interface A {
  void set(in int i);
}
interface B{
  void set(in float i);
}
class C implements A, B {
  void set[Int](in int i) from A.set;
  void set[Float](in float i) from B.set;
}
```
SIDL

Or change the name of an interface method that conflicts with your inherited class methods:

SIDL
```
interface A {
  void set(in int i);
}
class B {
  void set(in float i);
}
class C extends B implements A  {
  void set[Int](in int i) from A.set;
  void set(in float i); //Cannot use from on class methods
}
```

But it doesn't work for methods that have the same signature:

SIDL
```
/* X THIS WILL NOT COMPILE X */

interface A {
  void set(in int i);
}
interface B{
  void set(in int i);
}
class C implements A, B {
  void set[A](in int i) from A.set; //ERROR
  void set[B](in int i) from B.set; //signature conflict
}

/* X THIS WILL NOT COMPILE X */
```

5.3 Fundamental Types

Table 5.1 briefly shows the different data types that are supported in Babel. Refer to each chapter for the language specific bindings for each SIDL type. The "S" in SIDL stands for "Scientific." This emphasis is reflected in the fundamental support for complex numbers (`fcomplex` and `dcomplex`) and dynamic multidimensional arrays (`array<Type,Dim>`).

C++ developers looking at the SIDL syntax for arrays, might think that SIDL is a templated IDL, but this is not so. Although the syntax for SIDL arrays looks like a template, it is specific only to the array type. Developers cannot create templated classes or methods in SIDL.

> **Rationale:** *Although C++ templates are a very powerful programming mechanism, they apply only to C++. For Babel to implement similar hashing routines, method names in languages other than C++ would become prohibitively (thousands of characters) long. Moreover, this C++ template hashing mechanism is compiler specific so while C++ is very good at hiding the expanded template names (unless there is an error to report) we would have to add babel C++ bindings on a compiler by compiler basis.*

Discussion of the various types is broken up into sections. Numeric types such as `bool`, `char`, `int`, `long`, `float`, `double`, `fcomplex`, `dcomplex`, `string`s, as well as information about enumerated types and the opaque type are all covered in this Subsection 5.3.

Information about extended types such as Interfaces and Classes along with the methods they contain are described in Section 5.6, and Section 5.4 covers Array.

Table 5.1: SIDL Types

SIDL TYPE	SIZE (BITS)
bool	1
char	8
int	32
long	64
float	32
double	64
fcomplex	64
dcomplex	128
opaque	64
string	varies
enum	32
interface	varies
class	varies
array<Type,Dim>	varies
array< >	varies
rarray<Type,Dim> (*index variables*)	varies

Numeric Types

The SIDL types *bool*, *char*, *int*, *long*, *float*, *double*, *fcomplex*, and *dcomplex* are the smallest and easiest data types to transfer between languages transparently. They all have a fixed size and can just as reasonably be copied as passed by reference.

Most languages natively support all of these data types (though perhaps less so with complex types). There are a few notable exceptions that may be of interest.

ANSI C does not define the size of `int` and `long`, only that the latter be at least as big as the former. As of the C99 standard, there are types `int32_t` and `int64_t` that are signed integers that explicitly support a fixed number of bits. Most compilers already have these symbols defined appropriately in `sys/types.h` (pre C99 standard) or `inttypes.h`.

Python defines its `int` and `long` to be equivalent to C, and therefore suffers the same platform dependent integer size problem with less flexibility for a workaround. It is not uncommon for regression tests involving longs and Python to fail on certain platforms. Python 2.2 has a patch to make SIDL long support better.

Strings

Strings are an interesting datatype because they are fundamental to many pieces of software, but represented differently by practically every single programming language. Strings can have a high overhead to support language interoperability because there is invariably so much copying involved.

FORTRAN 77 and 90 support for strings is limited to a predetermined buffer size. Since the results of a string assignment into that buffer in FORTRAN does not propagate the length of the string, trailing whitespace is always trimmed for any string begin passed out from a FORTRAN implementation.

Opaque

The *opaque* type is dangerous and rarely useful. However, there are particular times when an opaque type is the only way to solve a problem; for example, it is one of the few portable ways to implement an object with state in Fortran 77 (see Section 8.8). When a SIDL file uses an *opaque* type, Babel guarantees only bits will be relayed exactly between caller and callee. If there is a need to pass more information than an opaque provides, than the developer can simply pass a pointer to that information.

Use of a *opaque* carries a heavy penalty. When Babel matures enough to support distributed computing, any

method calls with `opaque` in the argument list (or return type) will be restricted to in-process calls only.

> **Rationale:** *Since `opaque` is typically used for a pointer to memory, this sequence of bits has no meaning outside of its own process space.*

Enumerations

An enumeration is typically used in programming languages to specify a limited range of states to enable dealing with them by names instead of hard-coded values. For language interoperability purposes — especially to support this concept on languages with no native support — we've had to create specific rules for the integer values associated with enumerated types.

SIDL

```
package enumSample version 1.0 {

  // undefined integer values
  enum color {
    red, orange, yellow, green, blue, violet
  };

  // completely defined integer values
  enum car {
    /**
     * A sports car.
     */
    porsche = 911,
    /**
     * A family car.
     */
    ford = 150,
    /**
     * A luxury car.
     */
    mercedes = 550
  };

  // partially defined integer value
  enum number {
    notZero,     // This non-doc comment will not be retained.
    notOne,
    zero=0,
    one=1,
    negOne=-1,
    notNeg
  };
}
```

Above is a sample of enumerations taken directly from our regression tests. It defines a package `enumSample` that contains three enumerations. C/C++ developers will find the syntax very familiar. When defining an enumeration, the actual integer values assigned can be undefined, completely defined, or partially defined.

SIDL defines the following rules for adding integer values to enumerated states that don't have a value explicitly defined.

1. Error if two states are explicitly assigned the same value
2. Assign all explicit values to their named state.
3. Assign smallest unused non-negative value to first unassigned state in enumeration.

4. Repeat 3 until all states have assigned (unique) values.

To verify the application of these rules, the *enumSample.number* enumeration will have the following values assigned to its states: *NotZero*=2, *NotOne*=3, *zero*=0; *one*=1, *negOne*=-1, *notNeg*=4.

5.4 Arrays

One of the features that separates SIDL and BABEL from Microsoft's COM/DCOM and the OMG's CORBA is support for multi-dimensional arrays. SIDL is designed to serve the high performance computing community, so we anticipate that both SIDL object developers and object clients may require direct access to the underlying array data structure to try to optimize instruction pipelining or cache performance. The purpose of this document is to describe the functional API to the SIDL array data structure and the underlying data structures. This presentation will focus on the C API for arrays because it is the basis for the other language APIs, so they will likely reflect its idiosyncrasies.

There are two main kinds of arrays in SIDL: normal arrays and r-arrays. R-arrays are a specialized form of array for numeric types that has a simpler interface from C, C++, FORTRAN 77 and FOTRAN 90. Normal arrays are used for all SIDL types.

The SIDL array API and data structure can be used in client code to prepare argument for passing to a SIDL method, and it is used inside the implementation code to get data and meta-data from incoming arguments.

Normal SIDL arrays can be "row-major" or "column-major". They are not parallel array classes, and not particularly sophisticated, but they are very, very general. These are meant to generalize the array types built into many languages, not to provide a general array component that everyone will use. It is expected for parallel array libraries to build on top of the array type presented into SIDL.

R-arrays

There are two kinds of SIDL arrays: normal SIDL arrays and raw SIDL arrays called r-arrays. Normal SIDL arrays provide all the features of a normal SIDL type. They can be passed as `in`, `inout`, or `out` parameters, and they can be returned as a method return value. Normal SIDL arrays can be allocated or borrowed, and they are reference counted. You can also pass `NULL` as a normal SIDL array.

SIDL r-arrays exist to provide a lower access to numeric arrays from C, C++, Fortran 77, Fortran 90 and future languages as appropriate. For example, a one-dimensional r-array in C appears as a double pointer and a length parameter. To highlight the contrast, normal SIDL arrays appear as a struct in C, a template class in C++, an 64-bit integer in Fortran 77 and a derived type in Fortran 90.

R-arrays are have more restrictions in how they can be used. Here is how r-arrays are more constrainted:

1. Only the `in` and `inout` parameter modes are available for r-arrays. R-arrays cannot be used as return values or as `out` parameters.

2. R-arrays must be contiguous in memory, and multi-dimensional arrays must be in column-major order (i.e., Fortran order).

3. `NULL` is not an allowable value for an r-array parameter.

4. The semantics for `inout` r-array parameters are different. The implementation is not allowed to deallocate the array and return a new r-array. `inout` means that the array data is transferred from caller to callee at the start of a method invocation and from callee to caller at the end of the a method invocation.

5. The implementation of a method taking an r-array parameter cannot change the shape of the array.

6. The lower index is always 0, and the upper index is $n - 1$ where n is the length in a particular dimension. This is contrary to the normal convention for Fortran arrays.

7. It can only be used for arrays of SIDL `int`, `long`, `float`, `double`, `fcomplex`, and `dcomplex` types.

Rationale: *The way r-arrays are passed to the server-side code, particularly Fortran 77, makes it impossible for them to be allocated or deallocated. This makes* `out` *and return values impossible. Because the data has to be accessible directly from Fortran 77 without any additional meta-data, the array data must be in column-major order.*

Arrays of char are not currently supported for r-arrays because in some languages characters are treated as 16-bit Unicode characters.

The advantages of r-arrays include:

- Arrays appear more "natural" in C, C++, Fortran 77, Fortran 90 and future low level languages.
- Developers need less or no code to tranlate between their array data structure and SIDL's array data structure.
- SIDL generated APIs can have signatures very similar if not identical to well known legacy APIs.
- Less performance overhead because r-arrays can avoid a call to `malloc` and `free`.

When you declare an r-array, you also declare the index variables that will hold the size of the array in each dimension. For example, here is an method to solve one of the fundamental problems of linear algebra, $Ax = b$:

SIDL

```
void solve(in    rarray<double,2> A(m,n),
           inout rarray<double>   x(n),
           in    rarray<double>   b(m),
           in    int              m,
           in    int              n);
```

In this example, `A` is a 2-D array of doubles with `m` rows and `n` columns. `x` is a 1-D array of doubles of length `n`, and `b` is a 1-D array of doubles of length `m`. Note that by explicitly declaring the index variables, SIDL takes avoid using extra array size parameters by taking advantage of the fact that the sizes of `A`, `x` and `b` are all inter-related. The explicit declaration also allows the developer to control where the index parameters appear in the argument list. In many cases, the argument types and order can match existing APIs.

The mapping for the solve method will be shown for C, C++, Fortran 77 and Fortran 90 in the following chapters. In languages that do not support low level access such as Python and Java, r-arrays are treated just like normal SIDL arrays, and the redundant index arguments are dropped from the argument list. The indexing information is available from the SIDL array data structure.

SIDL Language Features

As of release 0.6.5, interface definitions can specify that an array argument or return value must have a particular ordering for a method. The type `array<int, 2, row-major>` indicates a dense, [1] two-dimensional array of 32 bit integers in row-major order; and likewise, the type `array<int, 2, column-major>` indicates an dense array in column-major order. Some numerical routines can only provide high performance with a particular type of array. The ordering is part of the interface definition to give clients the information they need to use the underlying code efficiently. The ordering specification is optional.

For one-dimensional arrays, specifying `row-major` or `column-major` allows you to specify that the array must be dense, that is stride 1. Otherwise, for one-dimensional arrays row-major and column-major are identical.

If you pass an array into a method and the array does not have the specified ordering, the skeleton code will make a copy of the array with the required ordering and pass the copy to the method. This copying is necessary for correctness, but it will cause a decrease in performance. The implementor of the method can count on an incoming array to have the required ordering.

For `out` parameters and return values, an ordering specification means that the method promises to return an array with the specified ordering. The implementation should create the `out` arrays with the proper ordering; because if it does not, the skeleton code will have to copy the outgoing array into a new array with the required ordering.

For `inout` parameters, an ordering specification means the ordering specification will be enforced by the skeleton code for the incoming and outgoing array value.

At the time of writing this, the ordering constraints are enforced for Python implementation because Python uses Numeric Python arrays, so BABEL cannot control the array ordering as fully. The Python skeletons do force outgoing arrays (i.e., arrays passed back from Python) to have the required ordering.

[1] meaning non-strided

Independent and borrowed arrays

From a memory perspective, there are two main kinds of arrays: independent and borrowed. The independent arrays owns and manages its data. It allocates space for the array elements when the array is created, and it deallocates that space when the array is finally destroyed.

The borrowed array does not own or manage its data. It borrows its array element data from another source that it cannot manage, and it only allocates space for the index bounds and stride information. The rationale for borrowed arrays is to allow data from another source to temporarily appear as a SIDL array without requiring data be copied.

If you `slice` an independent array, the resulting array is also considered independent even though it borrows data from the original independent array. The resulting array can still manage its data by retaining a reference to the original array; hence, its element data cannot disappear until the resulting array is destroyed. If you `slice` a borrowed array, the resulting array is also borrowed because like its original array, it doesn't manage the underlying data.

In the Babel generated code, r-arrays are converted to borrowed arrays. These borrowed arrays are allocated on the stack rather than on the heap to improve performance of r-arrays.

The Life of an Array

The existence of borrowed arrays causes the arrays to deviate from the normal reference counting pattern. You may recall that all arrays are reference counted, and an array's resources are reclaimed when the reference count goes to zero. However, a borrowed array's array element data will disappear whenever the source of the borrowed data determines that it should regardless of the reference count in corresponding the SIDL array. This behavior means that developers should consider any SIDL array that they did not create themselves, for example incoming arguments to methods, as potential borrowed arrays. When a method wants to keep a copy of an array that might be a borrowed array, it should use the `smartCopy` method documented below.

Here are some rules of thumb about the use of borrowed arrays:

- The creator of a borrowed array should guarantee that the data for the borrowed array will exist through the duration of any method calls using the borrowed array.
- Methods should not return a borrowed array as a return value or `out` parameter unless the method can guarantee that the array element data will be available until the process shuts down.
- There is a negligible performance cost when using `smartCopy` when the array is not borrowed, and there is a huge correctness benefit when the array is borrowed.

The Language Bindings

The C++ binding for array provides access to the C API in case you need to take the gloves off and revel in the data directly. But the C++ binding also provides a templated wrapper class to provide a more natural look and feel for C++ programmers.

In some cases, the Python binding for arrays must copy SIDL arrays to/from Numeric Python arrays; it should not happen for normally strided arrays except when an ordering constraint requires it. Arrays in Python don't have the SIDL methods available. They just have the Numeric Python API available.

The FORTRAN 77 API mimics the C API; all the C functions have been FORTRANified and have _f appended to their names. The FORTRAN 90 API uses function overloading to allow programmers to use the short array method names.

The Array API

In the following presentation, we use the SIDL *`double`* type; however, everything in this section applies to all types except where noted. The basic types are in the SIDL namespace. Table 5.2 shows the prefix for SIDL base types and the actual value type held by the array...

For arrays of interfaces or classes, the name of the array function prefix is derived from the fully qualified type name. For example, for the type *`sidl.BaseClass`*, the array functions all begin with `sidl_BaseClass`. For *`sidl.BaseInterface`*, they all begin with `sidl_BaseInterface`.

Table 5.2: SIDL types to array function prefixes

SIDL TYPE	ARRAY FUNCTION PREFIX	VALUE TYPE
bool	sidl_bool	sidl_bool
char	sidl_char	char
dcomplex	sidl_dcomplex	struct sidl_dcomplex
double	sidl_double	double
fcomplex	sidl_fcomplex	struct sidl_fcomplex
float	sidl_float	float
int	sidl_int	int32_t
long	sidl_long	int64_t
opaque	sidl_opaque	void *
string	sidl_string	char *

When you add an object or interface to an array, the reference count of the element being overwritten is decremented, and the reference count of the element being added is incremented. When you get an object or interface from an array, the caller owns the returned reference.

For arrays of strings when you add a string to any array, the array will store a copy of the string. When you retrieve a string from an array, you will receive a copy of the string. You should `sidl_String_free` the returned string when you are done with it.

When you create an array of interfaces, classes, or strings, all elements of the array are initialized to NULL. Other arrays are not initialized. When an array of interfaces, classes, or strings is destroyed, it releases any held references in the case of objects or interfaces. In the case of strings, it frees any non-NULL pointers.

The name of the data structure that holds the array if double is `struct sidl_double__array`. For some types, the data structure is an opaque type, and for others, it is defined in a public C header file.

The functions are listed succinctly in Table 5.3 as well as in detail over the next few pages.

Function: createCol

SIDL

```
/* C */
struct sidl_double__array*
sidl_double__array_createCol(int32_t       dimen,
                             const int32_t lower[],
                             const int32_t upper[]);
//
// C++
static sidl::array<double>
sidl::array<double>::createCol(int32_t       dimen,
                               const int32_t lower[],
                               const int32_t upper[]);
C
C FORTRAN 77
      subroutine sidl_double__array_createCol_f(dimen, lower, upper, result)
      integer*4 dimen
      integer*4 lower(dimen), upper(dimen)
      integer*8 result
!
! FORTRAN 90
subroutine createCol(lower, upper, result)
  integer (selected_int_kind(9)), dimension(:), intent(in) :: lower, upper
  type (sidl_double_3d), intent(out) :: result ! type depends on dimension
! dimension of result is inferred from the size of lower
```

Table 5.3: SIDL Array Functions

SHORT NAME	DESCRIPTION
createCol	Creates a column-major order SIDL array
createRow	Creates a row-major order SIDL array
create1d	Creates a dense one-dimensional SIDL array
create2dCol	Creates a dense, column-major, two-dimensional SIDL array
create2dRow	Creates a dense, column-major, two-dimensional SIDL array
slice	Creates a sub-array of another array. Takes parameters to define array properties.
borrow	Makes a SIDL array from thrid party data without copying it
smartCopy	Copies a borrowed array or addRefs a non-borrowed array
addRef	Increments the reference count.
deleteRef	Decrements the reference count.
get1	Returns the indexed element from a one-dimensional array
get2	Returns the indexed element from a two-dimensional array
get3	Returns the indexed element from a three-dimensional array
get4	Returns the indexed element from a four-dimensional array
get5	Returns the indexed element from a five-dimensional array
get6	Returns the indexed element from a six-dimensional array
get7	Returns the indexed element from a seven-dimensional array
get	Returns the indexed element from an array of any dimension
set1	Sets the indexed element in a one-dimensional array
set2	Sets the indexed element in a two-dimensional array
set3	Sets the indexed element in a three-dimensional array
set4	Sets the indexed element in a four-dimensional array
set5	Sets the indexed element in a five-dimensional array
set6	Sets the indexed element in a six-dimensional array
set7	Sets the indexed element in a seven-dimensional array
set	Sets the indexed element in an array of any dimension
dimen	Returns the dimension of the array
lower	Returns the lower bound of the specified dimension
upper	Returns the upper bound of the specified dimension
stride	Returns the stride of the specified dimension
length	Returns the length of the Array in the specified dimension
isColumnOrder	Returns true if the array is a dense column-major order array, false otherwise
isRowOrder	Returns true if the array is a dense row-major order array, false otherwise
copy	Copies the contents of source array to dest array
ensure	Returns an array with guaranteed ordering and dimension from any array.
first	Provides direct access to the element data of the array.

```
// Java
// (isRow should be false to get a column order array)
  public Array(int dim, int[] lower, int[] upper, boolean isRow);
```

This method creates a column-major, multi-dimensional array in a contiguous block of memory. `dimen` should be strictly greater than zero, and `lower` and `upper` should have `dimen` elements. `lower[i]` must be less than or equal to `upper[i]-1` for $i \geq 0$ and $i < dimen$. If this function fails for some reason, it returns `NULL`. `lower[i]` specifies the smallest valid index for dimension `i`, and `upper[i]` specifies the largest. Note this definition is somewhat un-C like where the upper bound is often one past the end. In SIDL, the size of dimension `i` is `1 + upper[i] - lower[i]`.

The function makes copies of the information provided by `dimen`, `lower`, and `upper`, so the caller is not obliged to maintain those values after the function call.

For FORTRAN, the new array is returned in the last parameter, `result`. A zero value in `result` indicates that the operation failed. For Fortran 90, you can use the function `not_null` to verify that `result` is a valid array.

Function: createRow

ANSI C

```
/* C */
struct sidl_double__array*
sidl_double__array_createRow(int32_t       dimen,
                             const int32_t lower[],
                             const int32_t upper[]);
//
// C++
static sidl::array<double>
sidl::array<double>::createRow(int32_t       dimen,
                               const int32_t lower[],
                               const int32_t upper[]);
C
C FORTRAN 77
       subroutine sidl_double__array_createRow_f(dimen, lower, upper, result)
       integer*4 dimen
       integer*4 lower(dimen), upper(dimen)
       integer*8 result
!
! FORTRAN 90
subroutine createRow(lower, upper, result)
  integer (selected_int_kind(9)), dimension(:), intent(in) :: lower, upper
  type(sidl_double_3d), intent(out) :: result ! type depends on dimension
! dimension of result is inferred from the size of lower

// Java
// (isRow should be true to get a row order array)
  public Array(int dim, int[] lower, int[] upper, boolean isRow);
```

This method creates a row-major, multi-dimensional array in a contiguous block of memory. Other than the difference in the ordering of the array elements, this method is identical to `createCol`.

Function: create1d

ANSI C

```
/* C */
struct sidl_double__array*
sidl_double__array_create1d(int32_t len);
```

```
// C++
static sidl::array<double>
sidl::array<double>::create1d(int32_t len);

C FORTRAN 77
       subroutine sidl_double__array_create1d_f(len, result)
       integer*4 len
       integer*8 result

! FORTRAN 90
subroutine create1d(len, result)
  integer (selected_int_kind(9)), intent(in) :: len
  type(sidl_double_1d), intent(out) :: result

// Java
  public Array1(int s0, boolean isRow);
```

This method creates a dense, one-dimensional vector of ints with a lower index of 0 and an upper index of $len - 1$. This is defined primarily as a convenience for C and C++ programmers; Fortran programmers should note that this subroutine creates arrays whose lower index is *0* not like standard Fortran arrays whose lower index is 1. If $len \leq 0$, this routine returns NULL.

Function: create2dCol

ANSI C

```
/* C */
struct sidl_double__array*
sidl_double__array_create2dCol(int32_t m, int32_t n);

// C++
static sidl::array<double>
sidl::array<double>::create2dCol(int32_t m, int32_t n);

C FORTRAN 77
       subroutine sidl_double__array_create2dCol_f(m, n, result)
       integer*4 m, n
       integer*8 result

! FORTRAN 90
subroutine create2dCol(m, n, result)
  integer (selected_int_kind(9)), intent(in) :: m, n
  type(sidl_double_2d), intent(out) :: result

// Java
// isRow should be false to get a column order array
  public Array2(int s0, int s1, boolean isRow);
```

This method creates a dense, column-major, two-dimensional array of ints with a lower index of $(0, 0)$ and an upper index of $(m - 1, n - 1)$. If $m \leq 0$ or $n \leq 0$, this method returns NULL. This is defined primarily as a convenience for C and C++ programmers; Fortran programmers should note that this subroutine creates arrays whose lower index is *0* not like standard Fortran arrays whose lower index is 1.

Function: create2dRow

ANSI C

```
/* C */
struct sidl_double__array*
```

```
sidl_double__array_create2dRow(int32_t m, int32_t n);

// C++
static sidl::array<double>
sidl::array<double>::create2dRow(int32_t m, int32_t n);

C FORTRAN 77
       subroutine sidl_double__array_create2dRow_f(m, n, result)
       integer*4 m, n
       integer*8 result

! FORTRAN 90
subroutine create2dRow(m, n, result)
  integer (selected_int_kind(9)), intent(in) :: m, n
  type(sidl_double_2d), intent(out) :: result

// Java
// isRow should be false to get a column order array
  public Array2(int s0, int s1, boolean isRow);
```

This method creates a dense, row-major, two-dimensional array of ints with a lower index of $(0, 0)$ and an upper index of $(m-1, n-1)$. If $m \leq 0$ or $n \leq 0$, this method returns NULL. This is defined primarily as a convenience for C and C++ programmers; Fortran programmers should note that this subroutine creates arrays whose lower index is *0* not like standard Fortran arrays whose lower index is 1.

Function: slice

ANSI C

```
/* C */
struct sidl_double__array *
sidl_double__array_slice(struct sidl_double__array *src,
                         int32_t                    dimen,
                         const int32_t              numElem[],
                         const int32_t             *srcStart,
                         const int32_t             *srcStride,
                         const int32_t             *newStart);
//
// C++
array<double>
sidl::array<double>::slice(int dimen,
                           const int32_t numElem[],
                           const int32_t *srcStart = 0,
                           const int32_t *srcStride = 0,
                           const int32_t *newStart = 0);
C
C FORTRAN 77
       subroutine sidl_double__array_slice_f(src, dimen, numElem, srcStart,
      $                        srcStride, newStart, result)
       integer*8 src, result
       integer*4 dimen
       integer*4 numElem(srcDimen), srcStart(srcDimen)
       integer*4 srcStride(srcDimen),  newStart(dimen)
!
! FORTRAN 90
subroutine slice(src, dimen, numElem, srcStart, srcStride, newStart, result)
  type(sidl_double_3d), intent(in) :: src     ! type depends on dimension
```

```
  type(sidl_double_2d), intent(out) :: result ! type depends on dimension
  integer (selected_int_kind(9)), intent(in) :: dimen
  integer (selected_int_kind(9)), intent(in), dimension(:) :: &
     numElem, srcStart, srcStride, newStart

// Java
  public native Array _slice(int dimen, int[] numElem, int[] srcStart,
                             int[] srcStride, int[] newStart);
```

This method will create a sub-array of another array. The resulting array shares data with the original array. The new array can be of the same dimension or potentially less than the original array. If you are removing a dimension, indicate the dimensions to remove by setting `numElem[i]` to zero for any dimension `i` that should go away in the new array. The meaning of each argument is covered below.

src the array to be created will be a subset of this array. If this argument is NULL, NULL will be returned. The returned array borrows data from `src`, so modifying one array modifies both. In C++, the `this` pointer takes the place of `src`.

dimen this argument must be greater than zero and less than or equal to the dimension of `src`. An illegal value will cause a NULL return value.

numElem this specifies how many elements from src should be in the new array in each dimension. A zero entry indicates that the dimension should not appear in the new array. This argument should be an array with an entry for each dimension of `src`. `NULL` will be returned for `src` if either

```
srcStart[i] + numElem[i] * srcStride[i]  >  upper[i], or
srcStart[i] + numElem[i] * srcStride[i]  <  lower[i]
```

srcStart this parameter specifies which element of `src` will be the first element of the new array. If this argument is NULL, the first element of `src` will be the first element of the new array. If non-NULL, this argument provides the coordinates of an element of `src`, so it must have an entry for each dimension of `src`. `NULL` will be returned for `src` if either

`srcStart[i] < lower[i]`, or `srcStart[i] > upper[i]`.

srcStride this argument lets you specify the stride between elements of `src` for each dimension. For example with a stride of 2, you could create a sub-array with only the odd or even elements of `src`. If this argument is NULL, the stride is taken to be one in each dimension. If non-NULL, this argument should be an array with an entry for each dimension of `src`.

newLower this argument is like the `lower` argument in a create method. It sets the coordinates for the first element in the new array. If this argument is NULL, the values indicated by `srcStart` will be used. If non-NULL, this should be an array with `dimen` elements.

Assuming the method is successful and the return value is named newArray, `src[srcStart]` refers to the same underlying element as `newArray[newStart]`.

If `src` is not a borrowed array (i.e., it manages its own data), the returned array can manage its by keeping a reference to `src`. It is not considered a borrowed array for purposes of `smartCopy`.

Function: borrow

ANSI C

```
/* C */
struct sidl_double__array*
sidl_double__array_borrow(double*       firstElement,
                          int32_t       dimen,
                          const int32_t lower[],
```

```
                            const int32_t upper[],
                            const int32_t stride[]);
//
// C++
void
sidl::array<double>::borrow(double*       firstElement,
                            int32_t       dimen,
                            const int32_t lower[],
                            const int32_t upper[],
                            const int32_t stride[]);
C
C FORTRAN 77
       subroutine sidl_double__array_borrow_f(firstElement, dimen, lower,
     $      upper, stride, result)
       real*8 firstElement()
       integer*4 dimen, lower(dimen), upper(dimen), stride(dimen)
       integer*8 result
!
! FORTRAN 90
subroutine borrow(firstElement, dimen, lower, upper, stride, &
                                       result)
  real (selected_real_kind(17,308)), intent(in) :: firstElement
  integer (selected_int_kind(9)), intent(in) :: dimen
  integer (selected_int_kind(9)), dimension(:), intent(in) :: lower, upper,&
                                                  stride
  type(sidl_double_1d), intent(out) :: result ! type depends on array dimension
```

This method creates a proxy SIDL multi-dimensional array using data provided by a third party. In some cases, this routine can be used to avoid making a copy of the array data. `dimen`, `lower`, and `upper` have the same meaning and constraints as in `SIDL_double__array_createCol`. The `firstElement` argument should be a pointer to the first element of the array; in this context, the first element is the one whose index is `lower`.

`stride[i]` specifies the signed offset from one element in dimension `i` to the next element in dimension `i`. For a one dimensional array, the first element has the address `firstElement`, the second element has the address `firstElement + stride[0]`, the third element has the address `firstElement + 2 * stride[0]`, etc. The algorithm for determining the address of the element in a multi-dimensional array whose index is in array `ind[]` is as follows:

ANSI C

```
int32_t* addr = firstElement;
for(int i = 0; i < dimen; ++i) {
  addr += (ind[i] - lower[i])*stride[i];
}
/* now addr is the address of element ind */
```

Note elements of stride need not be positive.

The function makes copies of the information provided by `dimen`, `lower`, `upper`, and `stride`. The type of `firstElement` is changed depending on the array value type (see Table 5.2).

Function: smartCopy

ANSI C

```
/* C */
struct sidl_double__array*
sidl_double__array_smartCopy(struct sidl_double__array *array);

// C++
void
```

```
sidl::array<double>::smartCopy();

C FORTRAN 77
       subroutine sidl_double__array_smartCopy_f(array, result)
       integer*8 array, result

! FORTRAN 90
subroutine smartCopy(array, result)
  type(sidl_double_1d), intent(in) :: array   ! type depends on dimension
  type(sidl_double_1d), intent(out) :: result ! type depends on dimension

// Java
  public native Array _smartCopy();
```

This method will copy a borrowed array or increment the reference count of an array that is able to manage its own data. This method is useful when you want to keep a copy of an incoming array. The C++ method operates on `this`.

Function: addRef

ANSI C

```
/* C */
void
sidl_double__array_addRef(struct sidl_double__array* array);

// C++
void
sidl::array<double>::addRef() throw ( NullIORException );

C FORTRAN 77
       subroutine sidl_double__array_addRef_f(array)
       integer*8 array

! FORTRAN 90
subroutine addRef(array)
  type(sidl_double_1d), intent(in) :: array ! type depends on array dimension
```

This increments the reference count by one. In C++, this method should be avoided because the C++ wrapper class manages the reference count for you.

Function: deleteRef

ANSI C

```
/* C */
void
sidl_double__array_deleteRef(struct sidl_double__array* array);

// C++
void
sidl::array<double>::deleteRef() throw ( NullIORException );

C FORTRAN 77
       subroutine sidl_double__array_deleteRef_f(array)
       integer*8 array

! FORTRAN 90
subroutine deleteRef(array)
  type(sidl_double_1d), intent(out) :: array ! type depends on dimension
```

This decreases the reference count by one. If this reduces the reference count to zero, the resources associated with the array are reclaimed. In C++, this method should be avoided because the C++ wrapper class manages the reference count for you.

Function: get1

ANSI C

```
/* C */
double
sidl_double__array_get1(const struct sidl_double__array* array,
                        int32_t                           i1);

// C++
double
sidl::array<double>::get(int32_t i1);

C FORTRAN 77
       subroutine sidl_double__array_get1_f(array, i1, result)
       integer*8 array
       integer*4 i1
       real*8 result

! FORTRAN 90
subroutine get(array, i1, result)
  type(sidl_int_1d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1
  real (selected_real_kind(17,308)), intent(out) :: result

// Java
  public double get(int i);
```

This method returns the element with index `i1` for a one dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 5.2). This method must only be called for one dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call free on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when i1 is out of bounds.

Function: get2

ANSI C

```
/* C */
double
sidl_double__array_get2(const struct sidl_double__array* array,
                        int32_t                           i1,
                        int32_t                           i2);

// C++
double
sidl::array<double>::get(int32_t i1, int32_t i2);

C FORTRAN 77
       subroutine sidl_int__array_get2_f(array, i1, i2, result)
       integer*8 array
       integer*4 i1, i2
       real*8 result
```

```
! FORTRAN 90
subroutine get(array, i1, i2, result)
  type(sidl_int_2d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2
  real (selected_real_kind(17,308)), intent(out) :: result

// Java
  public double get(int i, int j);
```

This method returns the element with indices (`i1`, `i2`) for a two dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 5.2. This method must only be called for two dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call free on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when `i1`, `i2` are out of bounds.

Function: get3

ANSI C

```
/* C */
double
sidl_double__array_get3(const struct sidl_double__array* array,
                        int32_t                           i1,
                        int32_t                           i2,
                        int32_t                           i3);
// C++
double
sidl::array<double>::get(int32_t i1, int32_t i2, int32_t i3);

C FORTRAN 77
      subroutine sidl_double__array_get3_f(array, i1, i2, i3, result)
      integer*8 array
      integer*4 i1, i2, i3
      real*8 result

! FORTRAN 90
subroutine get(array, i1, i2, i3, result)
  type(sidl_double_3d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2, i3
  real (selected_real_kind(17,308)), intent(out) :: result

// Java
  public double get(int i, int j, int k);
```

This method returns the element with indices (`i1`, `i2`, `i3`) for a three dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 5.2). This method must only be called for three dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call `free()` on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when i1, i2, i3 are out of bounds.

Function: get4

ANSI C

```
/* C */
double
sidl_double__array_get4(const struct sidl_double__array* array,
```

```
                      int32_t                         i1,
                      int32_t                         i2,
                      int32_t                         i3,
                      int32_t                         i4);
// C++
double
sidl::array<double>::get(int32_t i1, int32_t i2, int32_t i3, int32_t i4);

C FORTRAN 77
       subroutine sidl_double__array_get4_f(array, i1, i2, i3, i4, result)
       integer*8 array
       integer*4 i1, i2, i3, i4
       real*8 result

! FORTRAN 90
subroutine get(array, i1, i2, i3, i4, result)
  type(sidl_double_4d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2, i3, i4
  real (selected_real_kind(17,308)), intent(out) :: result

// Java
  public double get(int i, int j, int k, int l);
```

This method returns the element with indices(`i1`, `i2`, `i3`, `i4`) for a four dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 5.2). This method must only be called for four dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call `free()` on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when `i1`, `i2`, `i3`, or `i4` are out of bounds.

Function: get5-7

Methods `get5`–`get7` are defined in an analogous way.

Function: get

ANSI C

```
/* C */
double
sidl_double__array_get(const struct sidl_double__array* array,
                       const int32_t                     indices[]);

// C++
double
sidl::array<double>::get(const int32_t indices[]);

C FORTRAN 77
       subroutine sidl_double__array_get_f(array, indices, result)
       integer*8 array
       integer*4 indices()
       real*8 result

! FORTRAN 90
subroutine get(array, indices, result)
  type(sidl_real_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)), dimension(:), intent(in) ::indices
```

```
  real (selected_real_kind(17,308)), intent(out) :: result

// Java
  public native double _get(int i, int j, int k, int l, int m, int n, int o);
```

This method returns the element whose index is indices for an array of any dimension. The return type of this method is the value type for the SIDL type being held (see Table 5.2). This method can be called for any positively dimensioned array. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call `free()` on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when indices has an element out of bounds.

Function: set1

ANSI C

```
/* C */
void
sidl_double__array_set1(struct sidl_double__array* array,
                        int32_t                     i1,
                        double                      value));

// C++
void
sidl::array<int32_t>::set(int32_t i1, double value);

C FORTRAN 77
       subroutine sidl_double__array_set1_f(array, i1, value)
       integer*8 array
       integer*4 i1
       real*8 value

! FORTRAN 90
subroutine set(array, i1, value)
  type(sidl_double_1d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1,
  real (selected_real_kind(17,308)), intent(in) :: value

// Java
  public void set(int i, double value) {
```

This method sets the value in index `i1` of a one dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see Table 5.2). This method must only be called for one dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

Function: set2

ANSI C

```
/* C */
void
sidl_double__array_set2(struct sidl_double__array* array,
                        int32_t                     i1,
                        int32_t                     i2,
                        double                      value));

// C++
```

```
void
sidl::array<double>::set(int32_t i1, int32_t i2, double value);

C FORTRAN 77
       subroutine sidl_double__array_set2_f(array, i1, i2, value)
       integer*8 array
       integer*4 i1, i2
       real*8 value

! FORTRAN 90
subroutine set(array, i1, i2, value)
  type(sidl_int_2d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2
  real (selected_real_kind(17,308)), intent(in) :: value

// Java
  public void set(int i, int j, double value) {
```

This method sets the value in index (`i1`, `i2`) of a two dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see table 5.2). This method must only be called for two dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

Function: set3

ANSI C

```
/* C */
void
sidl_double__array_set3(struct sidl_double__array* array,
                        int32_t                     i1,
                        int32_t                     i2,
                        int32_t                     i3,
                        double                      value));

// C++
void
sidl::array<double>::set(int32_t i1, int32_t i2, int32_t i3, double value);

C FORTRAN 77
       subroutine sidl_double__array_set3_f(array, i1, i2, i3, value)
       integer*8 array
       integer*4 i1, i2, i3
       real*8 value

! FORTRAN 90
subroutine set(array, i1, i2, i3, value)
  type(sidl_double_3d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2, i3
  real (selected_real_kind(17,308)), intent(in) :: value

// Java
  public void set(int i, int j, int k, double value) {
```

This method sets the value in index (`i1`, `i2`, `i3`) of a three dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see table 5.2). This method must only be called for three

dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

Function: set4

ANSI C

```
/* C */
void
sidl_double__array_set4(struct sidl_double__array* array,
                        int32_t                     i1,
                        int32_t                     i2,
                        int32_t                     i3,
                        int32_t                     i4,
                        double                      value));
//
// C++
void
sidl::array<double>::set(int32_t i1, int32_t i2,
                         int32_t i3, int32_t i4, double value);
C
C FORTRAN 77
       subroutine sidl_double__array_set4_f(array, i1, i2, i3, i4, value)
       integer*8 array
       integer*4 i1, i2, i3, i4
       real*8 value
!
! FORTRAN 90
subroutine set(array, i1, i2, i3, i4, value)
  type(sidl_double_4d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2, i3, i4
  real (selected_real_kind(17,308)), intent(in) :: value

// Java
  public void set(int i, int j, int k, int l, double value) {
```

This method sets the value in index (`i1`, `i2`, `i3`, `i4`) of a four dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see table 5.2). This method must only be called for four dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

Function: set5-7

Methods `set5–set7` are defined in an analogous way.

Function: set

ANSI C

```
/* C */
void
sidl_double__array_set(struct sidl_double__array* array,
                       const int32_t               indices[],
                       double                      value);

// C++
void
```

```
sidl::array<double>::set(const int32_t indices[], double value);

C FORTRAN 77
       subroutine sidl_double__array_set_f(array, indices, value)
       integer*8 array
       integer*4 indices()
       real*8 value

! FORTRAN 90
subroutine set(array, indices, value)
  type(sidl_double_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)), intent(in), dimension(:) :: indices
  real (selected_real_kind(17,308)), intent(in) :: value

// Java
  public native void _set(int i, int j, int k, int l, int m, int n,
                          int o, double value);
```

This method sets the value in index indices for an array of any dimension to value. The type of the argument value is the value type for the SIDL type being held (see table 5.2). For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

Function: dimen

ANSI C

```
/* C */
int32_t
sidl_double__array_dimen(const struct sidl_double__array *array);

// C++
int32_t
sidl::array<double>::dimen() const;

C FORTRAN 77
       subroutine sidl_double__array_dimen_f(array, result)
       integer*8 array
       integer*4 result

! FORTRAN 90
integer (selected_int_kind(9)) dimen(array)
  type(sidl_double_1d) :: array ! type depends on dimension

// Java
public native int _dim();
```

This method returns the dimension of the array.

Function: lower

ANSI C

```
/* C */
int32_t
sidl_double__array_lower(const struct sidl_double__array *array, int32_t ind);

// C++
int32_t
```

```
sidl::array<double>::lower(int32_t ind) const;

C FORTRAN 77
       subroutine sidl_double__array_lower_f(array, ind, result)
       integer*8 array
       integer*4 ind, result

! FORTRAN 90
integer (selected_int_kind(9)) function lower(array, ind)
  type(sidl_double_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)) :: ind

// Java
  public native int _lower(int dim);
```

This method returns the lower bound on the index for dimension `ind` of array.

Function: upper

```
/* C */                                                              ANSI C
int32_t
sidl_double__array_upper(const struct sidl_double__array *array, int32_t ind);

// C++
int32_t
sidl::array<double>::upper(int32_t ind) const;

C FORTRAN 77
       subroutine sidl_double__array_upper_f(array, ind, result)
       integer*8 array
       integer*4 ind, result

! FORTRAN 90
integer (selected_int_kind(9)) function upper(array, ind)
  type(sidl_double_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)), intent(in) :: ind

// Java
  public native int _upper(int dim);
```

This method returns the upper bound on the index for dimension ind of array. If the upper bound is greater than or equal to the lower bound, the upper bound is a valid index (i.e., it is not one past the end).

Function: stride

```
/* C */                                                              ANSI C
int32_t
sidl_double__array_stride(const struct sidl_double__array *array, int32_t ind);

// C++
int32_t
sidl::array<double>::stride(int32_t ind) const;

C FORTRAN 77
       subroutine sidl_double__array_stride_f(array, ind, result)
```

```
       integer*8 array
       integer*4 ind, result

! FORTRAN 90
integer (selected_int_kind(9)) function stride(array, ind)
  type(sidl_double_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)) :: ind

// Java
  public native int _stride(int dim);
```

This method returns the stride for a particular dimension. This stride indicates how much to add to a pointer to get for the current element this the particular dimension to the next.

Function: length

ANSI C

```
/* C */
int32_t
sidl_double__array_length(const struct sidl_double__array *array, int32_t ind);

// C++ Default dimension is 1.
int32_t
sidl::array<int32_t>::length(int32_t ind = 0) const;

C FORTRAN 77
       subroutine sidl_double__array_length_f(array, ind, result)
       integer*8 array
       integer*4 ind, result

! FORTRAN 90
integer (selected_int_kind(9)) function length(array, ind)
  type(sidl_double_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)) :: ind

// Java
  public native int _length(int dim);

// For one dimensional Java arrays. Array1:
  public int lenth();
```

This method returns the length for a particular dimension. It is equivalent to the statement `upper(dim) - lower(dim) + 1`.

There is also a shortcut for one-dimensional arrays available in C++ and Java. In C++, if `length` is called with no arguments, it defaults to the first dimension. In Java `Array1` one-dimensional Java arrays have a length function that takes no arguments.

Function: isColumnOrder

ANSI C

```
/* C */
sidl_bool
sidl_double__array_isColumnOrder(const struct sidl_double__array *array);

// C++
bool
sidl::array<double>::isColumnOrder() const;
```

```
C FORTRAN 77
       subroutine sidl_double__array_isColumnOrder_f(array, result)
       integer*8 array
       logical   result

! FORTRAN 90
logical function isColumnOrder(array)
  type(sidl_double_2d), intent(in) :: array ! type depends on dimension

// Java
  public native boolean _isColumnOrder();
```

This method returns a true value if and only if `array` is dense, column-major ordered array. It does not modify the array at all.

Function: isRowOrder

ANSI C

```
/* C */
sidl_bool
sidl_double__array_isRowOrder(const struct sidl_double__array *array);

// C++
bool
sidl::array<double>::isRowOrder() const;

C FORTRAN 77
       subroutine sidl_double__array_isRowOrder_f(array, result)
       integer*8 array
       logical   result

! FORTRAN 90
logical function isRowOrder(array)
  type(sidl_double_1d), intent(int) :: array ! type depends on dimension

// Java
  public native boolean _isRowOrder();
```

This method returns a true value if and only if `array` is dense, row-major ordered array. It does not modify the array at all.

Function: copy

ANSI C

```
/* C */
void
sidl_double__array_copy(const struct sidl_double__array *src
                        struct sidl_double__array *dest);

// C++
void
sidl::array<double>::copy(const sidl::array<double> &src);

C FORTRAN 77
       subroutine sidl_double__array_copy_f(array, dest)
       integer*8 array, dest
```

```
! FORTRAN 90
subroutine copy(array, dest)
  type(sidl_double_1d), intent(in) :: array ! type depends on array dimension
  type(sidl_double_1d), intent(in) :: dest  ! type depends on array dimension

// Java
  public void _copy(sidl.Double.Array dest);
```

This method copies the contents of `src` to `dest`. For the copy to take place, both arrays must exist and be of the same dimension. This method will not modify `dest`'s size, index bounds, or stride; only the array element values of `dest` may be changed by this function. No part of `src` is changed by this method.

If `dest` has different index bounds than `src`, this method only copies the elements where the two arrays overlap. If `dest` and `src` have no indices in common, nothing is copied. For example, if `src` is a 1-d array with elements 0-5 and dest is a 1-d array with element 2-3, this function will copy element 2 and 3 from `src` to `dest`. If `dest` had elements 4-10, this method could copy elements 4 and 5.

Function: ensure

ANSI C

```
/* C */
struct sidl_double__array *
sidl_double__array_ensure(const struct sidl_double__array *src,
                          int32_t                          dimen,
                          int                              ordering);

// C++
void
sidl::array<double>::ensure(int32_t dimen, int ordering);

C FORTRAN 77
       subroutine sidl_double__array_ensure_f(src, dimen, ordering, result)
       integer*8 src, result
       integer*4 dimen, ordering

! FORTRAN 90
subroutine ensure(src, dimen, ordering, result)
   type(sidl_double_1d), intent(in) :: src    ! type depends on array dimension
   type(sidl_double_1d), intent(out) :: result! type depends on array dimension
   integer (selected_int_kind(9)) :: dimen, ordering
```

This method is used to obtain a matrix with a guaranteed ordering and dimension from an array with uncertain properties. If the incoming array has the required ordering and dimension, its reference count is incremented, and it is returned. If it doesn't, a copy with the correct ordering is created and returned. In either case, the caller knows that the returned matrix (if not NULL) has the desired properties.

This method is used internally to enforce the array ordering constraints in SIDL. Clients can use it in similar ways. However, because the method was intended as an internal Babel feature, is not available in Java or Python.

The ordering parameter should be one of the constants defined in `enum sidl_array_ordering` (e.g. `sidl_general_order`, `sidl_column_major_order`, or `sidl_row_major_order`). If you pass in `sidl_general_order`, this routine will only check the dimension of the matrix.

Function: first

ANSI C

```
/* C */
double *
sidl_double__array_first(const struct sidl_double__array *src);
```

```
// C++
double* first() throw();

C FORTRAN 77
      subroutine sidl_double__array_access_f(array, ref, lower, upper,
     $   stride, index)
      integer*8 array, index
      integer*4 lower(), upper(), stride()
      integer*4 ref()
```

This method provides direct access to the element data. Using this pointer and the stride information, you can perform your own array accesses without function calls. This method isn't available for arrays of strings, interface and objects because of memory/reference management issues. There is no equivalent of this of this function in Java or Python. To see how to get direct array access in FORTRAN 90, see Chapter 9.

The FORTRAN versions of the method return the lower, upper and stride information in three arrays, each with enough elements to hold an entry for each dimension of `array`. Because FORTRAN 77 does not have pointers, you must pass in a reference array, `array`. Upon exit, `ref(index)` is the first element of the array. The type of `ref` depends on the type of the array.

WARNING:

While calling the FORTRAN direct access routines, there is a possibility of an alignment error between your reference pointer, `ref`, and the pointer to the first element of the array data. The problem is more likely with arrays of `double` or `dcomplex`; although, it could occur with any type on some future platform. If `index` is zero on return, an alignment error occurred. If an alignment error occurs, you may be able to solve it by recompiling your FORTRAN files with flags to force doubles to be aligned on 8 byte boundaries. For example, the `-malign-double` flag for g77 forces doubles to be aligned on 64-bit boundaries. An alignment error occurs when `(char *)ref` minus `(char *)sidl_double__array_first(array)` is not integer divisible by `sizeof(datatype)` where `ref` refers to the address of the reference array.

Here is an example FORTRAN 77 subroutine to output each element of a 1-dimensional array of doubles using the direct access routine. FORTRAN 90 has a pointer in the array derived type when direct access is possible.

Fortran 77

```
C This subroutine will print each element of an array of doubles
      subroutine print_array(dblarray)
      implicit none
      integer*8 dblarray, index
      real*8 refarray(1)
      integer*4 lower(1), upper(1), stride(1), dimen, i
      if (dblarray .ne. 0) then
         call sidl_double__array_dimen_f(dblarray, dimen)
         if (dimen .eq. 1) then
            call sidl_double__array_access_f(dblarray, refarray,
     $           lower, upper, stride, index)
            if (index .ne. 0) then
               do i = lower(1), upper(1)
                  write(*,*) refarray(index + (i-lower(1))*stride(1))
               enddo
            else
               write(*,*) 'Alignment error occured'
            endif
         endif
```

```
      endif
      end
```

For a 2-dimensional array, the loop and array access is

Fortran 77

```
      do i = lower(1), upper(1)
         do j = lower(2), upper(2)
            write(*,*) refarray(index+(i-lower(1))*stride(1)+
     $           (j - lower(2))*stride(2))
         enddo
      enddo
```

Suppose you are wrapping a legacy FORTRAN application and you need to pass a SIDL array to a FORTRAN subroutine. Further suppose there is a FORTRAN 77 and FORTRAN 90 version of the subroutine. For example, the FORTRAN 77 subroutine has a signature such as:

Fortran 77

```
      subroutine TriedAndTrue(x, n)
      integer n
      real*8 x(n)
C insert wonderful, efficient, debugged code here
      end
```

The FORTRAN 90 subroutine has basically the same signature as follows:

Fortran 90

```
subroutine TriedAndTrue(x, n)
  integer (selected_int_kind(9))      :: n
  real (selected_real_kind(17, 308)) :: x(n)

  ! insert wonderful, efficient, debugged code here
end subroutine TriedAndTrue
```

Here is one way to wrap this method using SIDL. First of all, the SIDL method definition specifies that the array must be a 1-dimensional, column-major ordered array. This forces the incoming array to be a dense column.

SIDL

```
    static void TriedAndTrue(inout array<double,1,column-major> arg);
```

Given that method definition in a class named Class and a package named Pkg, the implementation of the wrapper should look something like the following for FORTRAN 77:

Fortran 77

```
        subroutine Pkg_Class_TriedAndTrue_fi(arg)
        implicit none
        integer*8 arg
C       DO-NOT-DELETE splicer.begin(Pkg.Class.TriedAndTrue)
        real*8 refarray(1)
        integer*4 lower(1), upper(1), stride(1)
        integer*8 index
        integer n
        call sidl_double__array_access_f(arg, refarray,
     $       lower, upper, stride, index)
        if (index .ne. 0) then
c we can assume stride(1) = 1 because of column-major specification
           n = 1 + upper(1) - lower(1)
           call TriedAndTrue(refarray(index), n)
        else
           write(*,*) 'ERROR: array alignment'
        endif
C       DO-NOT-DELETE splicer.end(Pkg.Class.TriedAndTrue)
        end
```

Similarly, it should look something like the following for FORTRAN 90, where the include statements are required at the top of the Impl file to ensure proper handling of subroutine names that have automatically been mangled by the Babel compiler:

Fortran 90

```
#include "Pkg_Class_fAbbrev.h"
#include "sidl_BaseClass_fAbbrev.h"
#include "sidl_BaseInterface_fAbbrev.h"
! DO-NOT-DELETE splicer.begin(_miscellaneous_code_start)
#include "sidl_double_fAbbrev.h"
! DO-NOT-DELETE splicer.end(_miscellaneous_code_start)
.
.
.
subroutine Pkg_Class_TriedAndTrue_mi(arg)
  ! DO-NOT-DELETE splicer.begin(Pkg.Class.TriedAndTrue.use)
  use SIDL_double_array
  ! DO-NOT-DELETE splicer.end(Pkg.Class.TriedAndTrue.use)
  implicit none
  type(sidl_double_a) :: arg

! DO-NOT-DELETE splicer.begin(Pkg.Class.TriedAndTrue)
  real (selected_real_kind(17,308)), dimension(1) :: refarray
  integer (selected_int_kind(8)), dimension(1)    :: low, up, str
  integer (selected_int_kind(8))    :: index, n
  call access(arg, refarray, low, up, str, index)
  if (index .ne. 0) then
    ! We can assume stride(1) = 1 because of column-major specification
    n = 1 + upper(1) - lower(1)
    call TriedAndTrue(refarray(index), n)
  else
    write(*,*) 'ERROR: array alignment'
  endif
! DO-NOT-DELETE splicer.end(Pkg.Class.TriedAndTrue)
end subroutine Pkg_Class_TriedAndTrue_mi
```

The C Macro API

Many of the SIDL array access functions have a corresponding C macro API for those who fear the function overhead of the C function API. When efficiency is not a concern, we recommend using the function API, but the C macro API is preferable to the direct access to the data structure. Parts of the macro API are not available for arrays of strings, interfaces or objects because the issues associated with memory and object reference management.

The macro API is very similar to the function API; however, a single set of macros applies to all the supported array types. The macro names are independent of the type of array you're accessing.

ANSI C

```
sidlArrayDim(array)
```

Return the dimension of array.

ANSI C

```
sidlLower(array,ind)
```

Return the lower bound on dimension ind.

ANSI C

```
sidlUpper(array,ind)
```

Return the upper bound on dimension ind.

```
sidlLength(array,ind)                                                ANSI C
```

Return the extent on dimension ind. The extent is equal to `sidlUpper(array,ind) - sidlLower(array,ind) + 1`.

```
sidlStride(array,ind)                                                ANSI C
```

Return the stride for dimension ind. The stride is the offset between elements in a particular dimension. It can be positive or negative. It is in terms of number of value types (i.e., it's 1 means contiguous regardless of what data type).

The macros to access array elements of array elements are unavailable for arrays of strings, classes and interfaces.

```
sidlArrayElem1(array, ind1)                                          ANSI C
sidlArrayElem2(array, ind1, ind2)
sidlArrayElem3(array, ind1, ind2, ind3)
sidlArrayElem4(array, ind1, ind2, ind3, ind4)
sidlArrayElem5(array, ind1, ind2, ind3, ind4, ind5)
sidlArrayElem6(array, ind1, ind2, ind3, ind4, ind5, ind6)
sidlArrayElem7(array, ind1, ind2, ind3, ind4, ind5, ind6, ind7)
```

Provide access to array elements to arrays of dimension 1–7. This macro can appear on the left hand side of an assignment or on the right hand side in an expression. These macros blindly assume that the dimension and indices are correct.

The macros to access the address of array elements are unavailable for arrays of strings, classes, and interfaces.

```
sidlArrayAddr1(array, ind1)                                          ANSI C
sidlArrayAddr2(array, ind1, ind2)
sidlArrayAddr3(array, ind1, ind2, ind3)
sidlArrayAddr4(array, ind1, ind2, ind3, ind4)
sidlArrayAddr5(array, ind1, ind2, ind3, ind4, ind5)
sidlArrayAddr6(array, ind1, ind2, ind3, ind4, ind5, ind6)
sidlArrayAddr7(array, ind1, ind2, ind3, ind4, ind5, ind6, ind7)
```

Return the address of elements in arrays of dimension 1–7. This macro can appear on the left hand side of an assignment or on the right hand side in an expression. These macros blindly assume that the dimension and indices are correct.

The C Data Structure

If even the macro interface is not fast enough for you, you can access the internal data structure for all the basic types except string. You cannot access the internal data structure for arrays of strings, interfaces and objects.

The basic form of the C data structure for type XXXX is:

```
struct sidl__array_vtable {                                          ANSI C

  /* Release resources associted with the array (refcount at zero) */
  void (*d_destroy)(struct sidl__array *);

  /* Clone or addRef depending on whether data is borrowed */
  struct sidl__array *(*d_smartcopy)(struct sidl__array *);

  /* Return the type of the array. */
  int32_t (*d_arraytype)(void);
};

struct sidl__array {
  int32_t                           *d_lower;
```

```
  int32_t                          *d_upper;
  int32_t                          *d_stride;
  const struct sidl__array_vtable *d_vtable;
  int32_t                           d_dimen;
  int32_t                           d_refcount;
};


struct sidl_XXXX__array {
  struct sidl__array       d_metadata;
  <value type for XXXX>   *d_firstElement;
};
```

The string "<value type for XXXX>" should be replaced by something like `sidl_bool`for an array of *bool*, `int32_t` for any array of *int*, `double` for an array of *double*, `int64_t` for an array of *long*, etc. (See Table 5.2)

d_dimen tells the dimension of the multi-dimensional array. `d_lower`, `d_upper`, and `d_stride` each point to arrays of `d_dimen int32_t`'s. `d_lower[i]` provides the lower bound for the index in dimension `i`, and `d_upper[i]` provides the upper bound for the index in dimension `i`. Both the lower and upper bounds are valid index values; the upper bound is not one past the end.

d_borrowed is true if the array does not managed the data that `d_firstElement` points too, and it is false otherwise. This mainly influences the behavior of the destructor.

Clients should not modify `d_lower`, `d_upper`, `d_stride`, `d_dimen`, `d_borrowed` or (in the case of pointers) the values to which they point.

d_stride[i] determines how elements are packed in dimension `i`. A value of 1 means that to get from element `j` to `j+1` in dimension `i`, you add one to the data pointer. Negative values for `d_stride` can be used to express a transposed matrix. The definition also allows either column or row major ordering for the data, and it also allows treating a subsection of an array as an array.

The data structure was inspired by the data structure used by Numeric Python; although, in Numeric Python, the stride is in terms of bytes. In SIDL, the stride is in terms of number of objects. One can convert to the Numeric Python view of things by multiplying the stride by the sizeof the value type.

Generic Arrays

The design of the array data structure enables the concept of a generic array, an array whose data type and dimension are unspecified. In SIDL, a generic array is indicated with the type *array< >*. There is no type or dimension information between the < and >.

Generic arrays are useful for making interfaces that are very flexible without requiring numerous methods to be defined. For example, if you were writing an interface to serialize an array, you could write one method *void serialize(in array< > array);* to handle an array of any type or dimension. Without generic arrays, you would have to define 77 different *serialize* methods to handle each possible array type and dimension.

In C, you can use the macro API to determine the dimension, bounds on each dimension and stride for a generic array. All other languages except Python provide a function API to determine the same information for a generic array. In Python, you determine the type and dimension of an array using the Numeric Python API.

The function API for generic arrays includes the following methods: `addRef`, `smartCopy`, `deleteRef`, `dimen`, `lower`, `upper`, `length`, `stride`, `isColumnOrder`, `isRowOrder`, and `type`. With the exception of `type`, these methods have all been presented above. The name of the method has the type left empty. Where the name for `addRef` in C on a double array is `sidl_double_array_addRef`, its name is `sidl__array_addRef` for a generic array. Note, there are two underscores between `sidl` and `array` in the generic array case.

The `type` method is defined as follows in the case of C.

ANSI C

```
/**
 * Return an integer indicating the type of elements held by the
```

```
 * array. Zero is returned if array is NULL.
 */
int32_t
sidl__array_type(const struct sidl__array* array);
```

It returns a value that indicates what the underlying type of the array actually is. The return value is either zero or one of the values in `sidl_array_type`.

ANSI C

```
enum sidl_array_type {
  /* these values must match values used in F77 & F90 too */
  sidl_bool_array      = 1,
  sidl_char_array      = 2,
  sidl_dcomplex_array  = 3,
  sidl_double_array    = 4,
  sidl_fcomplex_array  = 5,
  sidl_float_array     = 6,
  sidl_int_array       = 7,
  sidl_long_array      = 8,
  sidl_opaque_array    = 9,
  sidl_string_array    = 10,
  sidl_interface_array = 11 /* an array of sidl.BaseInterface's */
};
```

Once you've discovered the underlying type of the generic array, you can safely cast its pointer to the actual pointer type (in languages like C). Each language binding provides a way to cast generic array pointers to specific types and vice versa.

In the case of a `sidl_interface_array`, you can case the array to an array of *`sidl.BaseInterface`* interface references. Your code should treat it as such. You can downcast individual elements of the array as you need. Your code should consider the possibility that downcasting may fail. Babel can only guarantee that the elements of the array are *`sidl.BaseInterface`*'s.

5.5 SIDL Runtime

Inheritance

There is a collection of interfaces and classes that are defined by the SIDL runtime library. Some of these objects are implicitly inherited by objects and classes.

All classes that do not explicitly extend another class implicitly extend *`sidl.BaseClass`*. All interfaces that do not explicitly extend another interface implicitly extend *`sidl.BaseInterface`*. Furthermore, *`sidl.BaseClass`* implements *`sidl.BaseInterface`*. This means that all classes can be cast to a *`sidl.BaseClass`* and all objects can be cast to *`sidl.BaseInterface`*.

All exceptions must explicitly implement the interfaces in *`sidl.BaseException`*. The easiest way to do this is to is to extend the provided class *`sidl.SIDLException`*. This is a class the implements the basic Exception functionality for you, including `getNote` and `setNote`. You may also override one or more of these functions if you wish.

If a method in SIDL claims to throw an object that does not inherit from *`sidl.BaseException`*, this is an error and will be reported by Babel.

Interfaces

The SIDL runtime library provides three sets of interfaces:

Base The base class, interface, and exception upon which all Babel-enabled software builds.

Library Handler The DLL and Loader classes facilitate dynamic loading of objects at runtime.

Introspection The ClassInfo interface and ClassInfoI class enable checking meta-data associated with a class.

The interfaces for the runtime library, as described in SIDL, are:

SIDL

```
//
// File:        sidl.sidl
// Revision:    @(#) $Revision: 1262 $
// Date:        $Date: 2005-12-12 16:36:46 -0800 (Mon, 12 Dec 2005) $
// Description: sidl interface description for the basic sidl run-time library
//
// Copyright (c) 2001, The Regents of the University of Calfornia.
// Produced at the Lawrence Livermore National Laboratory.
// Written by the Components Team <components@llnl.gov>
// UCRL-CODE-2002-054
// All rights reserved.
//
// This file is part of Babel. For more information, see
// http://www.llnl.gov/CASC/components/. Please read the COPYRIGHT file
// for Our Notice and the LICENSE file for the GNU Lesser General Public
// License.
//
// This program is free software; you can redistribute it and/or modify it
// under the terms of the GNU Lesser General Public License (as published by
// the Free Software Foundation) version 2.1 dated February 1999.
//
// This program is distributed in the hope that it will be useful, but
// WITHOUT ANY WARRANTY; without even the IMPLIED WARRANTY OF
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the terms and
// conditions of the GNU Lesser General Public License for more details.
//
// You should have recieved a copy of the GNU Lesser General Public License
// along with this program; if not, write to the Free Software Foundation,
// Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

/**
 * The <code>sidl</code> package contains the fundamental type and interface
 * definitions for the <code>sidl</code> interface definition language.  It
 * defines common run-time libraries and common base classes and interfaces.
 * Every interface implicitly inherits from <code>sidl.BaseInterface</code>
 * and every class implicitly inherits from <code>sidl.BaseClass</code>.
 *
 */
package sidl version 0.9.12 {

  /**
   * Every interface in <code>sidl</code> implicitly inherits
   * from <code>BaseInterface</code>, and it is implemented
   * by <code>BaseClass</code> below.
   */
  interface BaseInterface {

    /**
     * <p>
     * Add one to the intrinsic reference count in the underlying object.
     * Object in <code>sidl</code> have an intrinsic reference count.
```

```
    * Objects continue to exist as long as the reference count is
    * positive. Clients should call this method whenever they
    * create another ongoing reference to an object or interface.
    * </p>
    * <p>
    * This does not have a return value because there is no language
    * independent type that can refer to an interface or a
    * class.
    * </p>
    */
  void addRef();

  /**
    * Decrease by one the intrinsic reference count in the underlying
    * object, and delete the object if the reference is non-positive.
    * Objects in <code>sidl</code> have an intrinsic reference count.
    * Clients should call this method whenever they remove a
    * reference to an object or interface.
    */
  void deleteRef();

  /**
    * Return true if and only if <code>obj</code> refers to the same
    * object as this object.
    */
  bool isSame(in BaseInterface iobj);

  /**
    * Return whether this object is an instance of the specified type.
    * The string name must be the <code>sidl</code> type name.  This
    * routine will return <code>true</code> if and only if a cast to
    * the string type name would succeed.
    */
  bool isType(in string name);

  /**
    * Return the meta-data about the class implementing this interface.
    */
  ClassInfo getClassInfo();
}

/**
 * Every class implicitly inherits from <code>BaseClass</code>.  This
 * class implements the methods in <code>BaseInterface</code>.
 */
class BaseClass implements BaseInterface {
  /**
    * <p>
    * Add one to the intrinsic reference count in the underlying object.
    * Object in <code>sidl</code> have an intrinsic reference count.
    * Objects continue to exist as long as the reference count is
    * positive. Clients should call this method whenever they
    * create another ongoing reference to an object or interface.
    * </p>
```

```
    * <p>
    * This does not have a return value because there is no language
    * independent type that can refer to an interface or a
    * class.
    * </p>
    */
   final void addRef();

   /**
    * Decrease by one the intrinsic reference count in the underlying
    * object, and delete the object if the reference is non-positive.
    * Objects in <code>sidl</code> have an intrinsic reference count.
    * Clients should call this method whenever they remove a
    * reference to an object or interface.
    */
   final void deleteRef();

   /**
    * Return true if and only if <code>obj</code> refers to the same
    * object as this object.
    */
   final bool isSame(in BaseInterface iobj);

   /**
    * Return whether this object is an instance of the specified type.
    * The string name must be the <code>sidl</code> type name.  This
    * routine will return <code>true</code> if and only if a cast to
    * the string type name would succeed.
    */
   bool isType(in string name);

   /**
    * Return the meta-data about the class implementing this interface.
    */
   final ClassInfo getClassInfo();
}

/**
 * This package has some I/O capability that's not core to the
 * SIDL object model, but still needed by parts of the generated code
 */
package io {
  /**
   *    Objects that implement Serializable will be serializable (copyable)
   * over RMI, or storable to streams.  Classes that can pack or unpack
   *  themselves should implement this interface
   */
   interface Serializable {
     void packObj( in Serializer ser );
     void unpackObj( in Deserializer des );
   }
}

/**
```

```
 * Every exception implements <code>BaseException</code>. This interface
 * declares the basic functionality to get and set error messages and stack
 * traces.
 */
interface BaseException extends sidl.io.Serializable{

  /**
   * Return the message associated with the exception.
   */
  string getNote();

  /**
   * Set the message associated with the exception.
   */
  void setNote(in string message);

  /**
   * Returns formatted string containing the concatenation of all
   * tracelines.
   */
  string getTrace();

  /**
   * Adds a stringified entry/line to the stack trace.
   */
  void add[Line](in string traceline);

  /**
   * Formats and adds an entry to the stack trace based on the
   * file name, line number, and method name.
   */
  void add(in string filename, in int lineno, in string methodname);
}

/**
 * This exception type is the default exception for every method.
 *
 */
interface RuntimeException extends BaseException {}

/**
 * <code>SIDLException</code> provides the basic functionality of the
 * <code>BaseException</code> interface for getting and setting error
 * messages and stack traces.
 */
class SIDLException implements-all BaseException {
}

/**
 * <code>PreViolation</code> provides the basic marker for
 * a pre-condition exception.
 */
class PreViolation extends SIDLException implements RuntimeException {
}
```

```
/**
 * <code>PostViolation</code> provides the basic marker for
 * a post-condition exception.
 */
class PostViolation extends SIDLException implements RuntimeException {
}

/**
 * <code>InvViolation</code> provides the basic marker for
 * a invariant exception.
 */
class InvViolation extends SIDLException implements RuntimeException {
}

/**
 * When loading a dynamically linked library, there are three
 * settings: LOCAL, GLOBAL and SCLSCOPE.
 */
enum Scope {
   /** Attempt to load the symbols into a local namespace. */
   LOCAL,
   /** Attempt to load the symbols into the global namespace. */
   GLOBAL,
   /** Use the scope setting from the SCL file. */
   SCLSCOPE
}

/**
 * When loading a dynmaically linked library, there are three
 * settings: LAZY, NOW, SCLRESOLVE
 */
 enum Resolve {
   /** Resolve symbols on an as needed basis. */
   LAZY,
   /** Resolve all symbols at load time. */
   NOW,
   /** Use the resolve setting from the SCL file. */
   SCLRESOLVE
 }

/**
 * The <code>DLL</code> class encapsulates access to a single
 * dynamically linked library.  DLLs are loaded at run-time using
 * the <code>loadLibrary</code> method and later unloaded using
 * <code>unloadLibrary</code>.  Symbols in a loaded library are
 * resolved to an opaque pointer by method <code>lookupSymbol</code>.
 * Class instances are created by <code>createClass</code>.
 */
class DLL {

  /**
   * Load a dynamic link library using the specified URI.  The
   * URI may be of the form "main:", "lib:", "file:", "ftp:", or
```

```
 * "http:".  A URI that starts with any other protocol string
 * is assumed to be a file name.  The "main:" URI creates a
 * library that allows access to global symbols in the running
 * program's main address space.  The "lib:X" URI converts the
 * library "X" into a platform-specific name (e.g., libX.so) and
 * loads that library.  The "file:" URI opens the DLL from the
 * specified file path.  The "ftp:" and "http:" URIs copy the
 * specified library from the remote site into a local temporary
 * file and open that file.  This method returns true if the
 * DLL was loaded successfully and false otherwise.  Note that
 * the "ftp:" and "http:" protocols are valid only if the W3C
 * WWW library is available.
 *
 * @param uri          the URI to load. This can be a .la file
 *                     (a metadata file produced by libtool) or
 *                     a shared library binary (i.e., .so,
 *                     .dll or whatever is appropriate for your
 *                     OS)
 * @param loadGlobally <code>true</code> means that the shared
 *                     library symbols will be loaded into the
 *                     global namespace; <code>false</code>
 *                     means they will be loaded into a
 *                     private namespace. Some operating systems
 *                     may not be able to honor the value presented
 *                     here.
 * @param loadLazy     <code>true</code> instructs the loader to
 *                     that symbols can be resolved as needed (lazy)
 *                     instead of requiring everything to be resolved
 *                     now (at load time).
 */
bool loadLibrary(in string uri,
                 in bool loadGlobally,
                 in bool loadLazy);

/**
 * Get the library name.  This is the name used to load the
 * library in <code>loadLibrary</code> except that all file names
 * contain the "file:" protocol.
 */
string getName();

/**
 * Unload the dynamic link library.  The library may no longer
 * be used to access symbol names.  When the library is actually
 * unloaded from the memory image depends on details of the operating
 * system.
 */
void unloadLibrary();

/**
 * Lookup a symbol from the DLL and return the associated pointer.
 * A null value is returned if the name does not exist.
 */
opaque lookupSymbol(in string linker_name);
```

```
  /**
   * Create an instance of the sidl class.  If the class constructor
   * is not defined in this DLL, then return null.
   */
  BaseClass createClass(in string sidl_name);
}

/**
 * Interface <code>Finder</code> is an interface for classes that resolve
 * dynamic libraries.
 * Class <code>Loader</code> takes one of these interfaces through the
 * method <code>setFinder</code>.  If NULL is passed to setFinder, the
 * class <code>DefaultFinder</code> is used.
 */
interface Finder {
  /**
   * Find a DLL containing the specified information for a sidl
   * class. This method searches through the files in set set path
   * looking for a shared library that contains the client-side or IOR
   * for a particular sidl class.
   *
   * @param sidl_name  the fully qualified (long) name of the
   *                   class/interface to be found. Package names
   *                   are separated by period characters from each
   *                   other and the class/interface name.
   * @param target     to find a client-side binding, this is
   *                   normally the name of the language.
   *                   To find the implementation of a class
   *                   in order to make one, you should pass
   *                   the string "ior/impl" here.
   * @param lScope     this specifies whether the symbols should
   *                   be loaded into the global scope, a local
   *                   scope, or use the setting in the file.
   * @param lResolve   this specifies whether symbols should be
   *                   resolved as needed (LAZY), completely
   *                   resolved at load time (NOW), or use the
   *                   setting from the file.
   * @return a non-NULL object means the search was successful.
   *         The DLL has already been added.
   */
  DLL findLibrary(in string  sidl_name,
                  in string  target,
                  in Scope   lScope,
                  in Resolve lResolve);

  /**
   * Set the search path, which is a semi-colon separated sequence of
   * URIs as described in class <code>DLL</code>.  This method will
   * invalidate any existing search path.
   */
  void setSearchPath(in string path_name);

  /**
```

```
    * Return the current search path.  If the search path has not been
    * set, then the search path will be taken from environment variable
    * SIDL_DLL_PATH.
    */
   string getSearchPath();

   /**
    * Append the specified path fragment to the beginning of the
    * current search path.  If the search path has not yet been set
    * by a call to <code>setSearchPath</code>, then this fragment will
    * be appended to the path in environment variable SIDL_DLL_PATH.
    */
   void addSearchPath(in string path_fragment);

}

/**
 *  This class is the Default Finder.  If no Finder is set in class Loader,
 *  this finder is used.  It uses SCL files from the filesystem to
 *  resolve dynamic libraries.
 *
 * The initial search path is taken from the SIDL_DLL_PATH
 * environment variable.
 */

class DFinder implements-all Finder {
}


/**
 * Class <code>Loader</code> manages dyanamic loading and symbol name
 * resolution for the sidl runtime system.  The <code>Loader</code> class
 * manages a library search path and keeps a record of all libraries
 * loaded through this interface, including the initial "global" symbols
 * in the main program.
 *
 * Unless explicitly set, the <code>Loader</code> uses the default
 * <code>sidl.Finder</code> implemented in <code>sidl.DFinder</code>.
 * This class searches the filesystem for <code>.scl</code> files when
 * trying to find a class. The initial path is taken from the
 * environment variable SIDL_DLL_PATH, which is a semi-colon
 * separated sequence of URIs as described in class <code>DLL</code>.
 */
class Loader {

   /**
    * Load the specified library if it has not already been loaded.
    * The URI format is defined in class <code>DLL</code>.  The search
    * path is not searched to resolve the library name.
    *
    * @param uri          the URI to load. This can be a .la file
    *                     (a metadata file produced by libtool) or
    *                     a shared library binary (i.e., .so,
    *                     .dll or whatever is appropriate for your
```

```
     *                       OS)
     * @param loadGlobally <code>true</code> means that the shared
     *                       library symbols will be loaded into the
     *                       global namespace; <code>false</code>
     *                       means they will be loaded into a
     *                       private namespace. Some operating systems
     *                       may not be able to honor the value presented
     *                       here.
     * @param loadLazy     <code>true</code> instructs the loader to
     *                       that symbols can be resolved as needed (lazy)
     *                       instead of requiring everything to be resolved
     *                       now.
     * @return if the load was successful, a non-NULL DLL object is returned.
     */
    static DLL loadLibrary(in string uri,
                           in bool loadGlobally,
                           in bool loadLazy);

    /**
     * Append the specified DLL to the beginning of the list of already
     * loaded DLLs.
     */
    static void addDLL(in DLL dll);

    /**
     * Unload all dynamic link libraries.  The library may no longer
     * be used to access symbol names.  When the library is actually
     * unloaded from the memory image depends on details of the operating
     * system.
     */
    static void unloadLibraries();

    /**
     * Find a DLL containing the specified information for a sidl
     * class. This method searches SCL files in the search path looking
     * for a shared library that contains the client-side or IOR
     * for a particular sidl class.
     *
     * This call is implemented by calling the current
     * <code>Finder</code>. The default finder searches the local
     * file system for <code>.scl</code> files to locate the
     * target class/interface.
     *
     * @param sidl_name  the fully qualified (long) name of the
     *                     class/interface to be found. Package names
     *                     are separated by period characters from each
     *                     other and the class/interface name.
     * @param target       to find a client-side binding, this is
     *                     normally the name of the language.
     *                     To find the implementation of a class
     *                     in order to make one, you should pass
     *                     the string "ior/impl" here.
     * @param lScope       this specifies whether the symbols should
     *                     be loaded into the global scope, a local
```

```
     *                   scope, or use the setting in the SCL file.
     * @param lResolve   this specifies whether symbols should be
     *                   resolved as needed (LAZY), completely
     *                   resolved at load time (NOW), or use the
     *                   setting from the SCL file.
     * @return a non-NULL object means the search was successful.
     *         The DLL has already been added.
     */
    static DLL findLibrary(in string  sidl_name,
                           in string  target,
                           in Scope   lScope,
                           in Resolve lResolve);

    /**
     * Set the search path, which is a semi-colon separated sequence of
     * URIs as described in class <code>DLL</code>.  This method will
     * invalidate any existing search path.
     *
     * This updates the search path in the current <code>Finder</code>.
     */
    static void setSearchPath(in string path_name);

    /**
     * Return the current search path.  The default
     * <code>Finder</code> initializes the search path
     * from environment variable SIDL_DLL_PATH.
     *
     */
    static string getSearchPath();

    /**
     * Append the specified path fragment to the beginning of the
     * current search path.  This method operates on the Loader's
     * current <code>Finder</code>. This will add a path to the
     * current search path. Normally, the search path is initialized
     * from the SIDL_DLL_PATH environment variable.
     */
    static void addSearchPath(in string path_fragment);

    /**
     * This method sets the <code>Finder</code> that
     * <code>Loader</code> will use to find DLLs.  If no
     * <code>Finder</code> is set or if NULL is passed in, the Default
     * Finder <code>DFinder</code> will be used.
     *
     * Future calls to <code>findLibrary</code>,
     * <code>addSearchPath</code>, <code>getSearchPath</code>, and
     * <code>setSearchPath</code> are deligated to the
     * <code>Finder</code> set here.
     */
    static void setFinder(in Finder f);

    /**
     * This method gets the <code>Finder</code> that <code>Loader</code>
```

```
    * uses to find DLLs.
    */
  static Finder getFinder();
}

/**
 * This provides an interface to the meta-data available on the
 * class.
 */
interface ClassInfo {
   /**
    * Return the name of the class.
    */
   string getName();

  /**
   * Get the version of the intermediate object representation.
   * This will be in the form of major_version.minor_version.
   */
   string getIORVersion();
}

/**
 * An implementation of the <code>ClassInfo</code> interface. This
 * provides methods to set all the attributes that are read-only in
 * the <code>ClassInfo</code> interface.
 */
class ClassInfoI implements-all ClassInfo {
   /**
    * Set the name of the class.
    */
   final void setName(in string name);

   /**
    * Set the IOR major and minor version numbers.
    */
   final void setIORVersion(in int major, in int minor);
}

/**
 * Exception thrown from Babel internals when memory allocation
 * fails
 */
class MemoryAllocationException extends sidl.SIDLException
  implements RuntimeException {
}

/**
 * Exception is thrown when a cast fails and the failure needs to
 * be communicated up the call stack.  (Note: babel _cast does NOT
 * throw this exception)
 */
class CastException extends sidl.SIDLException
   implements RuntimeException {
```

```
}

/**
 * This Exception is thrown by the Babel runtime when a non SIDL
 * exception is thrown from an exception throwing language such as
 * C++ or Java.
 */
class LangSpecificException extends sidl.SIDLException
    implements RuntimeException {
}

/**
 * This package has some I/O capability that's not core to the SIDL
 * object model, but still needed by parts of the generated code
 */
package io {

  /** generic exception for I/O issues */
  class IOException extends sidl.SIDLException
     implements RuntimeException {
  }


  /**
   * Standard interface for packing Babel types
   */
  interface Serializer {
    void packBool( in string key, in bool value )
       throws IOException ;
    void packChar( in string key, in char value )
       throws IOException ;
    void packInt( in string key, in int value )
       throws IOException ;
    void packLong( in string key, in long value )
       throws IOException ;
    void packOpaque(  in string key, in opaque value )
       throws IOException ;
    void packFloat( in string key, in float value )
       throws IOException ;
    void packDouble( in string key, in double value )
       throws IOException ;
    void packFcomplex( in string key, in fcomplex value )
       throws IOException ;
    void packDcomplex( in string key, in dcomplex value )
       throws IOException ;
    void packString( in string key, in string value )
       throws IOException ;
    void packSerializable( in string key, in Serializable value )
       throws IOException;

    /**
     * pack arrays of values.  It is possible to ensure an array is
     * in a certain order by passing in ordering and dimension
     * requirements.  ordering should represent a value in the
```

```
 * sidl_array_ordering enumeration in sidlArray.h If either
 * argument is 0, it means there is no restriction on that
 * aspect.  The boolean reuse_array flag is set to true if the
 * remote unserializer should try to reuse the array that is
 * passed into it or not.
 */
void packBoolArray( in string key, in array<bool> value,
                    in int ordering, in int dimen,
                    in bool reuse_array )
   throws IOException ;
void packCharArray( in string key, in array<char> value,
                    in int ordering, in int dimen,
                    in bool reuse_array )
   throws IOException ;
void packIntArray( in string key, in array<int> value,
                   in int ordering, in int dimen,
                   in bool reuse_array )
   throws IOException ;
void packLongArray( in string key, in array<long> value,
                    in int ordering, in int dimen,
                    in bool reuse_array )
   throws IOException ;
void packOpaqueArray( in string key, in array<opaque> value,
                      in int ordering, in int dimen,
                      in bool reuse_array )
   throws IOException ;
void packFloatArray( in string key, in array<float> value,
                     in int ordering, in int dimen,
                     in bool reuse_array )
   throws IOException ;
void packDoubleArray( in string key, in array<double> value,
                      in int ordering, in int dimen,
                      in bool reuse_array )
   throws IOException ;
void packFcomplexArray( in string key, in array<fcomplex> value,
                        in int ordering, in int dimen,
                        in bool reuse_array )
   throws IOException ;
void packDcomplexArray( in string key, in array<dcomplex> value,
                        in int ordering, in int dimen,
                        in bool reuse_array )
   throws IOException ;
void packStringArray( in string key, in array<string> value,
                      in int ordering, in int dimen,
                      in bool reuse_array )
   throws IOException ;
void packGenericArray( in string key, in array<> value,
                       in bool reuse_array )
   throws IOException ;
void packSerializableArray( in string key,
                            in array<Serializable> value,
                            in int ordering, in int dimen,
                            in bool reuse_array )
   throws IOException;
```

```
}

/**
 * Standard interface for unpacking Babel types
 */
interface Deserializer {
  /* unpack values */
  void unpackBool( in string key, out bool value )
     throws IOException ;
  void unpackChar( in string key, out char value )
     throws IOException ;
  void unpackInt( in string key, out int value )
     throws IOException ;
  void unpackLong( in string key, out long value )
     throws IOException ;
  void unpackOpaque( in string key, out opaque value )
     throws IOException ;
  void unpackFloat( in string key, out float value )
     throws IOException ;
  void unpackDouble( in string key, out double value )
     throws IOException ;
  void unpackFcomplex( in string key, out fcomplex value )
     throws IOException ;
  void unpackDcomplex( in string key, out dcomplex value )
     throws IOException ;
  void unpackString( in string key, out string value )
     throws IOException ;
  void unpackSerializable( in string key, out Serializable value )
     throws IOException;

  /** unpack arrays of values
   * It is possible to ensure an array is
   * in a certain order by passing in ordering and dimension
   * requirements.  ordering should represent a value in the
   * sidl_array_ordering enumeration in sidlArray.h If either
   * argument is 0, it means there is no restriction on that
   * aspect.  The rarray flag should be set if the array being
   * passed in is actually an rarray.  The semantics are slightly
   * different for rarrays.  The passed in array MUST be reused,
   * even if the array has changed bounds.
   */
  void unpackBoolArray( in string key, out array<bool> value,
                        in int ordering, in int dimen,
                        in bool isRarray )
     throws IOException ;
  void unpackCharArray( in string key, out array<char> value,
                        in int ordering, in int dimen,
                        in bool isRarray  )
     throws IOException ;
  void unpackIntArray( in string key, out array<int> value,
                       in int ordering, in int dimen,
                       in bool isRarray  )
     throws IOException ;
  void unpackLongArray( in string key, out array<long> value,
```

```
                          in int ordering, in int dimen,
                          in bool isRarray  )
        throws IOException ;
    void unpackOpaqueArray( in string key, out array<opaque> value,
                            in int ordering, in int dimen,
                            in bool isRarray  )
        throws IOException ;
    void unpackFloatArray( in string key, out array<float> value,
                           in int ordering, in int dimen,
                           in bool isRarray  )
        throws IOException ;
    void unpackDoubleArray( in string key, out array<double> value,
                            in int ordering, in int dimen,
                            in bool isRarray  )
        throws IOException ;
    void unpackFcomplexArray( in string key, out array<fcomplex> value,
                              in int ordering, in int dimen,
                              in bool isRarray  )
        throws IOException ;
    void unpackDcomplexArray( in string key,
                              out array<dcomplex> value,
                              in int ordering, in int dimen,
                              in bool isRarray  )
        throws IOException ;
    void unpackStringArray( in string key, out array<string> value,
                            in int ordering, in int dimen,
                            in bool isRarray  )
        throws IOException ;
    void unpackGenericArray( in string key, out array<> value)
        throws IOException ;
    void unpackSerializableArray( in string key,
                                  out array<Serializable> value,
                                  in int ordering, in int dimen,
                                  in bool isRarray  )
        throws IOException ;
  }

} //end package io

/**
 * This package contains necessary interfaces for RMI protocols to
 * hook into Babel, plus a Protocol Factory class.  The intention is
 * that authors of new protocols will create classes that implement
 * InstanceHandle, Invocation and Response (they could even have one
 * object that implements all three interfaces).
 */
package rmi {

  /**
   * Generic Network Exception
   */
   class NetworkException extends sidl.io.IOException {
      int getHopCount();
  }
```

```
/**
 * This exception is thrown by the RMI library when a
 * host can not be found by a DNS lookup.
 */
class UnknownHostException extends NetworkException {}

/**
 * This exception is normally thrown by the RMI library when the
 * server is started up and the port it is assigned to use is
 * already in use.
 */
class BindException extends NetworkException {}

/**
 * This exception is thrown by the RMI library when an
 * attempt to connect to a remote host fails.
 */
class ConnectException extends NetworkException {}

/**
 * This exception is thrown by the RMI library when a host
 * can be found by DNS, but is not reachable.  It usually means
 * a router is down.
 */
class NoRouteToHostException extends NetworkException {}

/**
 * This exception is thrown by the RMI library when a request
 * times out.
 */
class TimeOutException extends NetworkException {}

/**
 * This exception is thrown by the RMI library when the network
 * unexpected loses it's connection.  Can be caused by reset,
 * software connection abort, connection reset by peer, etc.
 */
class UnexpectedCloseException extends NetworkException {}

/**
 * This exception is thrown by a server when a passed in object
 * id does not match any known object.
 */
class ObjectDoesNotExistException extends NetworkException {}

/**
 * This exception is thrown by the RMI library when a passed in URL
 * is malformed.
 */
class MalformedURLException extends NetworkException {}

/**
 * This is a base class for all protocol specific exceptions.
```

```
 */
 class ProtocolException extends NetworkException {}

/**
 * This singleton class keeps a table of string prefixes
 * (e.g. "babel" or "proteus") to protocol implementations.  The
 * intent is to parse a URL (e.g. "babel://server:port/class") and
 * create classes that implement
 * <code>sidl.rmi.InstanceHandle</code>.
 */
class ProtocolFactory {
  /**
   * Associate a particular prefix in the URL to a typeName
   * <code>sidl.Loader</code> can find.  The actual type is
   * expected to implement <code>sidl.rmi.InstanceHandle</code>
   * Return true iff the addition is successful.  (no collisions
   * allowed)
   */
   static bool addProtocol( in string prefix, in string typeName )
       throws NetworkException;

  /**
   * Return the typeName associated with a particular prefix.
   * Return empty string if the prefix
   */
   static string getProtocol( in string prefix )
      throws NetworkException;

  /**
   * Remove a protocol from the active list.
   */
   static bool deleteProtocol( in string prefix )
      throws NetworkException;

  /**
   * Create a new remote object and return an instance handle for that
   * object.
   * The server and port number are in the url.  Return nil
   * if protocol unknown or InstanceHandle.init() failed.
   */
  static InstanceHandle createInstance( in string url,
                                        in string typeName )
       throws NetworkException;

  /**
   * Create an new connection linked to an already existing
   * object on a remote server.  The server and port number are in
   * the url, the objectID is the unique ID of the remote object
   * in the remote instance registry.  Return null if protocol
   * unknown or InstanceHandle.init() failed.  The boolean addRef
   * should be true if connect should remotely addRef
   */
   static InstanceHandle connectInstance( in string url, in bool ar)
       throws NetworkException;
```

```
  }

/**
 * This interface holds the state information for handles to
 * remote objects.  Client-side messaging libraries are expected
 * to implement <code>sidl.rmi.InstanceHandle</code>,
 * <code>sidl.rmi.Invocation</code> and
 * <code>sidl.rmi.Response</code>.
 *
 * Every stub with a connection to a remote object holds a pointer
 * to an InstanceHandle that manages the connection. Multiple
 * stubs may point to the same InstanceHandle, however.  Babel
 * takes care of the reference counting, but the developer should
 * keep concurrency issues in mind.
 *
 *  When a new remote object is created:
 *       sidl_rmi_InstanceHandle c =
 *         sidl_rmi_ProtocolFactory_createInstance( url, typeName,
 *               _ex );
 *
 *  When a new stub is created to connect to an existing remote
 * instance:
 *        sidl_rmi_InstanceHandle c =
 *          sidl_rmi_ProtocolFactory_connectInstance( url, _ex );
 *
 *  When a method is invoked:
 *       sidl_rmi_Invocation i =
 *          sidl_rmi_InstanceHandle_createInvocation( methodname );
 *       sidl_rmi_Invocation_packDouble( i, "input_val" , 2.0 );
 *       sidl_rmi_Invocation_packString( i, "input_str", "Hello" );
 *       ...
 *       sidl_rmi_Response r = sidl_rmi_Invocation_invokeMethod( i );
 *       sidl_rmi_Response_unpackBool( i, "_retval", &succeeded );
 *       sidl_rmi_Response_unpackFloat( i, "output_val", &f );
 *
 */
interface InstanceHandle {

  /** initialize a connection (intended for use by the
   * ProtocolFactory, (see above).  This should parse the url and
   * do everything necessary to create the remote object.
   */
   bool initCreate( in string url, in string typeName )
       throws NetworkException;

  /**
   * initialize a connection (intended for use by the ProtocolFactory)
   * This should parse the url and do everything necessary to connect
   * to a remote object.
   */
   bool initConnect( in string url, in bool ar) throws NetworkException;

  /** return the short name of the protocol */
   string getProtocol() throws NetworkException;
```

```
  /** return the object ID for the remote object*/
   string getObjectID() throws NetworkException;

  /**
   * return the full URL for this object, takes the form:
   * protocol://serviceID/objectID (where serviceID would = server:port
   * on TCP/IP)
   * So usually, like this: protocol://server:port/objectID
   */
  string getObjectURL() throws NetworkException;

  /** create a serializer handle to invoke the named method */
  Invocation createInvocation( in string methodName )
      throws NetworkException;

  /**
   * closes the connection (called by the destructor, if not done
   * explicitly) returns true if successful, false otherwise
   * (including subsequent calls)
   */
  bool close() throws NetworkException;
}

/**
 * This type is used to pack arguments and make the Client->Server
 * method invocation.
 */
interface Invocation extends sidl.io.Serializer {

  /**
   * this method is one of a triad.  Only one of which
   * may be called, and it must the the last method called
   * in the object's lifetime.
   */
  Response invokeMethod() throws NetworkException;

  /**
   * This method is second of the triad.  It returns
   * a Ticket, from which a Response is later extracted.
   */
  Ticket invokeNonblocking() throws NetworkException;

  /**
   * This method is third of the triad.  It returns
   * and exception iff the invocation cannot be delivered
   * reliably.  It does not wait for the invocation to
   * be acted upon and returns no values from the invocation.
   */
  void invokeOneWay() throws NetworkException;
}

/**
 * This type is created when an invokeMethod is called on an
```

```
 * Invocation.  It encapsulates all the results that users will
 * want to pull out of a remote method invocation.
 */
interface Response extends sidl.io.Deserializer {

  /**
   * May return a communication exception or an execption thrown
   * from the remote server.  If it returns null, then it's safe
   * to unpack arguments
   */
  sidl.BaseException getExceptionThrown() throws NetworkException;

}

/**
 * This interface is implemented by the Server side deserializer.
 * Deserializes method arguments in preperation for the method
 * call.
 */
interface Call extends sidl.io.Deserializer { }

/**
 * This interface is implemented by the Server side serializer.
 * Serializes method arguments after the return from the method
 * call.
 */
interface Return extends sidl.io.Serializer {

  /**
   * This method serialized exceptions thrown on the server side
   * that should be returned to the client.  Assumed to invalidate
   * in previously serialized arguments.  (Also assumed that no
   * more arguments will be serialized.)
   */
  void throwException(in sidl.BaseException ex_to_throw)
    throws NetworkException;
}

/**
 * Used in lieu of a Response in nonblocking calls
 */
interface Ticket {

    /** blocks until the Response is recieved */
    void block();

    /**
     * returns immediately: true iff the Response is already
     * received */
    bool test();

    /** creates an empty container specialized for Tickets */
    TicketBook createEmptyTicketBook();
```

```
    /** returns immediately: returns Response or null
     *  (NOTE: needed for implementors of communication
     *         libraries, not expected for general use).
     */
    Response getResponse() throws NetworkException;

}

/**
 * This is a collection of Tickets that itself can be viewed
 * as a ticket.
 */
interface TicketBook extends Ticket {

    /** insert a ticket with a user-specified ID */
    void insertWithID( in Ticket t, in int id );

    /** insert a ticket and issue a unique ID */
    int insert( in Ticket t );

    /** remove a ready ticket from the TicketBook
     *  returns 0 (and null) on an empty TicketBook
     */
    int removeReady( out Ticket t );

    /**
     * immediate, returns the number of Tickets in the book.
     */
    bool isEmpty();

}

/**
 * This singleton class is implemented by Babel's runtime for RMI
 * libraries to invoke methods on server objects.  It maps
 * objectID strings to sidl_BaseClass objects and vice-versa.
 *
 * The InstanceRegistry creates and returns a unique string when a
 * new object is added to the registry.  When an object's refcount
 * reaches 0 and it is collected, it is removed from the Instance
 * Registry.
 *
 * Objects are added to the registry in 3 ways:
 * 1) Added to the server's registry when an object is
 *    create[Remote]'d.
 * 2) Implicity added to the local registry when an object is
 *    passed as an argument in a remote call.
 * 3) A user may manually add a reference to the local registry
 *    for publishing purposes.  The user hsould keep a reference
 *    to the object.  Currently, the user cannot provide their own
 *    objectID, this capability should probably be added.
 */
class InstanceRegistry {
```

```
    /**
     * Register an instance of a class.
     *
     * the registry will return an objectID string guaranteed to be
     * unique for the lifetime of the process
     */
    static string registerInstance( in sidl.BaseClass instance )
        throws NetworkException;

    /**
     * returns a handle to the class based on the unique objectID
     * string, (null if the handle isn't in the table)
     */
    static sidl.BaseClass getInstanceString( in string instanceID )
        throws NetworkException;

    /**
     * takes a class and returns the objectID string associated
     * with it.  (null if the handle isn't in the table)
     */
    static string getInstanceClass( in sidl.BaseClass instance )
        throws NetworkException;

    /**
     * removes an instance from the table based on its objectID
     * string..  returns a pointer to the object, which must be
     * destroyed.
     */
    static sidl.BaseClass removeInstanceString( in string instanceID )
        throws NetworkException;

    /**
     * removes an instance from the table based on its BaseClass
     * pointer.  returns the objectID string, which much be freed.
     */
    static string removeInstanceClass( in sidl.BaseClass instance )
        throws NetworkException;
}

/**
 * This singleton class is implemented by Babel's runtime for to
 * allow RMI downcasting of objects.  When we downcast an RMI
 * object, we may be required to create a new derived class object
 * with a connect function.  We store all the connect functions in
 * this table for easy access.
 *
 * This Class is for Babel internal use only.
 */
class ConnectRegistry {

    /**
     * The key is the SIDL classname the registered connect belongs
     * to.  Multiple registrations under the same key are possible,
     * this must be protected against in the user code.  Babel does
```

```
     * this internally with a static boolean.
     */
    static void registerConnect( in string key, in opaque func);

    /**
     * Returns the connect method for the class named in the key
     */
    static opaque getConnect( in string key );

    /**
     * Returns the connect method for the class named in the key,
     * and removes it from the table.
     */
    static opaque removeConnect( in string key );
}

/**
 * ServerInfo is an interface (possibly implemented by the ORB
 * itself) that provides functions to deal with the problems
 * associated with passing local object remotely.  It should be
 * registered with the ServerRegistry for general use.
 */
interface ServerInfo {
    string getServerURL(in string objID) throws NetworkException;

    /**
     * For internal Babel use ONLY. Needed by Babel to determine if
     * a url points to a local or remote object.  Returns the
     * objectID if is local, Null otherwise.
     */
    string isLocalObject(in string url) throws NetworkException;

    /**
     * This gets an array of logged exceptions.  If an exception
     * can not be thrown back to the caller, we log it with the
     * Server.  This gets the array of all those exceptions.  THIS
     * IS SOMETHING OF A TEST! THIS MAY CHANGE!
     */
    array<sidl.io.Serializable,1> getExceptions();
}

/**
 * This singleton class is simply a place to register a
 * ServerInfo interface for general access.  This ServerInfo
 * should give info about the ORB being used to export RMI objects
 * for the current Babel process.
 *
 * This Registry provides two important functions, a way to get
 * the URL for local object we wish to expose over RMI, and a way
 * to tell if an object passed to this process via RMI is actually
 * a local object.  This abilities are protocol specific, the
 * ServerInfo interface must by implemented by the protocol
 * writer.
 *
```

```
     * THIS CLASS IS NOT DESIGNED FOR CONCURRENT WRITE ACCESS.  (Only
     * one server is assumed per Babel process)
     */
    class ServerRegistry {
      static void registerServer(in sidl.rmi.ServerInfo si)
          throws NetworkException;

      /**
       * Perhaps this should take BaseClass and look the objectID up in
       * the Instance Registry
       */
      static string getServerURL(in string objID) throws NetworkException;

      /**
       * For internal Babel use ONLY. Needed by Babel to determine if a
       * url points to a local or remote object.  Returns the objectID
       * if is local, Null otherwise.
       */
      static string isLocalObject(in string url) throws NetworkException;

      /**
       * This gets an array of logged exceptions.  If an exception
       * can not be thrown back to the caller, we log it with the
       * Server.  This gets the array of all those exceptions.  THIS
       * IS SOMETHING OF A TEST! THIS MAY CHANGE!
       */
      static array<sidl.io.Serializable,1> getExceptions();

    }

    /** This is the basic interface that all RMI servers should implement.*/
    interface BaseServer {
      /**
       * The url string is necessary to give the server the info it
       * needs to start up.  The string is protocol dependant.  In
       * most TCP/IP based protocols, only the port number is really
       * required, but it is suggested they accept a full url of the
       * form: protocol://localhost:port
       */
      void init(in string url) throws sidl.rmi.NetworkException;

      /**
       * Starts the server running.  It is expected that most servers
       * will be threaded, and this function returns a long that can
       * be used as a server thread id to join to is necessary.
       */
      long run() throws sidl.rmi.NetworkException;
    }

  } //end package rmi
}
```

5.6 Objects

One of the strategies that SIDL uses to enforce language interoperability is to define an object model that it supports across all language bindings. This enables real object-oriented programming in non OO languages such as C and FORTRAN 77. This also means that the inheritance mechanisms inside real OO languages may be circumvented.

Contrary to newer scripting languages such as Python and Ruby, not everything in SIDL is an object. Only classes (abstract or not) and interfaces are objects. Everything else (e.g. arrays, enums, strings, ints) is something other than an object and therefore outside the scope of this Section.

Babel's Object Model

SIDL defines three types of objects: interfaces, classes, and abstract classes. A SIDL *interface* is akin to a Java interface or a C++ pure abstract base class. It is an object that defines methods (aka member functions), but carries no implementation of those methods. A *class* by comparison is always concrete; meaning that there is an implementation for each of its methods and it can be instantiated. An *abstract class* falls somewhere between an *interface* and a *class*. It has at least one method unimplemented, so it cannot be instantiated, but it also may have several methods that are implemented and these implementations can be inherited.

SIDL supports multiple inheritance of interfaces and single inheritance of implementation. This is a strategy found in other OO languages such as Java and ObjectiveC. The words to distinguish these two forms of inheritance are *extends* and *implements*. Interfaces can extend multiple interfaces, but they cannot implement anything. Classes can extend at most one other class (abstract or not), but can implement multiple interfaces.

Furthermore, any inherited abstract methods (inherited from either and abstract parent class or and implemented interface) will default to abstract unless they are re-declared in the current class. If a concrete class implements many large interfaces, this can result in a fairly large list of redeclared functions in the class definition. As a shortcut, we included the `implements-all` directive, a short hand that states explicitly that we intend to implement every method in the named interface concretely. That's why, in the following example, class B must be declared abstract, but class D is concrete. Class B does not redeclare the `printMe` function, but class D `implements-all`. There is no similar directive for inheritance from abstract classes.

We display a small SIDL file below and finish this Subsection with a discussion of its details.

SIDL

```
package object version 1.0 {

  interface A {
    void display();
    void printMe();
  }

  abstract class B implements A {
    void display();
  }

  class C extends B {
    void printMe();
  }

  class D implements-all A {
  }
}
```

object.A is an interface that has two methods *display()* and *print()*. Both of these methods take no arguments and return no value. (We will discuss arguments and return values in the next section.) Since *object.A* is an interface, there is no implementation associated with it, and Babel will not generate any implementation code associated with it.

object.B is an abstract class that inherits from *object.A*. Since it redeclares the *display()* method, Babel will generate the appropriate code for an implementation of this method only. It will not generate code for the other

inherited method *print()* (since it wasn't declared in the SIDL file) and it will not generate constructors/destructors since the class is abstract.

object.C is a concrete class that extends the abstract class *object.B* it then lists only the unimplemented method *print()*, implying that it will use the implementation of *display()* it inherited from its parent.

object.D is also a concrete class that uses the *implements-all* directive. This is identical to using *implements* and then listing all the methods declared in the interface. The *implements-all* directive was added to SIDL as a convenience construct and to save excessive typing in the SIDL file. By virtue of the *implements-all* directive, *object.D* will provide its own implementation of all of *object.A*'s methods, namely *display()* and *print()*.

Methods on Objects

Methods in SIDL are virtual by default. This means that the actual binding of a method invocation to an actual implementation is determined at runtime, based on the concrete type of the object.

SIDL currently defines three modifiers to methods that change their default behavior.

- *final* : Final methods are the opposite of virtual. While they may still be inherited by child classes, they cannot be overridden.
- *static* : Static methods are sometimes called "class methods" because they are part of a class, but do not depend on an object instance. In non-OO languages, this means that the typical first argument of an instance is removed. In OO languages, these are mapped directly to an Java or C++ static method.
- *local* : local is a keyword that relates to RMI. A local method cannot be called on a remote object. Any call on a local method must be an in-process call, or a PreViolation will be thrown.
- *oneway* : oneway is a keyword that relates to RMI. A oneway method can only take in arguments, no out arguments. This allows the oneway method to be called with a oneway network message, so the user doesn't need to wait for a response.
- *nonblocking* : nonblocking is a keyword that relates to RMI. A nonblocking method is split into two methods, method_send() and method_recv(). method_send() takes the in arguments and immediately returns a sidl.rmi.Ticket. This Ticket can be used to determine when the remote method returns, and get the out arguemnts.

Starting with Babel `0.11.0`, all SIDL methods implicitly throw *sidl.RuntimeException*. A *sidl.RuntimeException* can be generated by the Babel generated glue code. For example, if the code is making a call across the network using remote method invocation and the network goes down, Babel's glue code would generate a *RuntimeException*. In cases where the implementation throws an unexpected exception (i.e., not one that is declared in the method's SIDL declaration), the glue code can generate a *RuntimeException*.

Parameter Passing

Each parameter in a method call obeys the following syntax

```
[ (modifier) ] (mode) (type) (name)
```

Where `(mode)` is one of *in*, *out*, or *inout*; `(type)` is any SIDL recognized type; and `(name)` is any non-reserved word[2]. The `(modifier)` is optional, and currently unimplemented. SIDL currently reserves the word *copy* for future use as an parameter modifier, and may add others in the future[3].

For new users, the parameter's mode (e.g. *in*, *out*, or *inout*) is perhaps the most troublesome. On the surface, it's easy to explain that *in* parameters are passed into the code, *out* parameters come out, and *inout* parameters do both. More specifically the rules are:

1. *in* does not mean `const`.

[2] Refer to Section A.2 for the list of reserved words

[3] Babel is still pre-1.0 after all!

2. *in* arguments are passed by value, therefore what happens inside the function has no effect on the value passed in (from the perspective of the caller).

3. *inout* arguments are passed by reference. The callee is allowed to do whatever it wants with the data passed in, and changes made by the callee are sent back to the caller. For interfaces, classes, and normal arrays, the callee can even destroy the reference, create a new object or array, and return a reference to it.

4. Objects, interfaces and arrays should be allocated using the create methods provided. Types created on the stack should never be passed as an *inout* argument, since the implementation may want to destroy it.

5. *out* arguments are also passed by reference, but the incoming value is ignore and typically overwritten. *Do Not* attempt to pass in a value to a function through an out argument. There is no guarantee that the data will make it to the Implementation, and if the data is lost, there is no guarantee the reference will be correctly destroyed.

Method Overloading

Method overloading is the object-oriented practice of defining more than one method with the same name in a class. Doing so allows the convenient reuse of a method name when, for example, the underlying implementations differ based on the types of the arguments. Actually, support for overloaded methods typically relies on the signature of each method to ensure uniqueness. In this case, the signature consists of the method name along with the number, types, and ordering of its arguments.

Since Babel supports languages that do not support method overloading, a mechanism for generating unique names was needed. These are typically generated by compilers based on hashing the argument types into the method name. However, developers often manually address this with far fewer characters than would be used by a compiler. Consequently, it was determined it would be more efficient to leave the task of identifying the unique name to the developer. Therefore, Babel allows the specification of the base, or short, method name along with an optional method name extension as illustrated in the SIDL file below for the `getValue` method.

SIDL

```
package Overload version 1.0 {

  class Sample {
    int       getValue ( );
    int       getValue[Int]( in int v );
    double    getValue[Double]( in double v );
  }
}
```

Thus, the full method name is the concatenation of the short name followed by the name extension. When generating code for supported languages, Babel makes use of either the short or full method name as appropriate for the language(s) involved. For those that support method overloading, such as C++ and Java, Babel relies only on the short method name, thus ignoring the extension. For the rest, like C, Fortran, and Python, Babel must make use of the full name to ensure methods are uniquely identified.

In the example above, the first method specification takes no arguments so has no name extension. This is acceptable because there are no potentially conflicting methods at this point for any programming language supported by Babel. The second method, with the user-defined name extension of `Int`, takes a single int argument, resulting in the unique method name `getValueInt`. The last method, with a user-defined name extension of `Double`, takes a single double argument, resulting in the unique method name of `getValueDouble`. Examples of calling overloaded methods from Babel-supported languages can be found in the respective language binding chapters.

5.7 XML Repositories

Even though SIDL is currently the primary input format for Babel, it is not the only format Babel understands. For type repositories (similar in function to include directories for C/C++ headers) the preferred language to articulate types is XML.

Babel has the capabilities to convert SIDL files into XML files adhering to the `SIDL.dtd`. This capability is explained further in Chapter 13. The XML files in these repositories can be included in subsequent runs quickly since all the external references were resolved by Babel during their creation. A SIDL file may refer to unresolved types.

Part II

Supported Language Bindings

Chapter 6

C Bindings

Contents

6.1 Introduction

This chapter provides an introduction to the C bindings for SIDL. Babel supports both callers and callees written in C so this chapter illustrates the use of Babel for both. That is, it shows how to use Babel to wrap the implementation of software written in C as well as how to call software, possibly implemented in any other supported language, from C.

Since Babel's Intermediate Object Representation (IOR) is written in C, the C bindings are very similar to the IOR. In addition, all of the objects in the *sidl* namespace (e.g. *sidl.BaseClass*, etc.) are implemented in C, so clients can develop solely with a C compiler if necessary. Of course this seems a little silly since the intent of Babel is to provide multilingual interoperability.

6.2 Basic Types

The basic types in SIDL are mapped into C according to Table 6.1.

6.3 Header files

If you would like to use type `X.Y.Z` from C (package X, subpackage Y, class Z), you should `#include "X_Y_Z.h"` . If you would like to include the header files for a whole package `X.Y`, you can `#include "X_Y.h"`. For example, you can include all the types in the `sidl` namespace with `#include "sidl.h"`.

Each client side header file will ensure that `sidl_header.h` is included. `sidl_header.h` defines:

1. `struct sidl_dcomplex` for the SIDL dcomplex type with parts named `real` and `imaginary`;
2. `struct sidl_fcomplex` for the SIDL fcomplex type with parts named `real` and `imaginary`;

Table 6.1: SIDL to C Type Mappings

SIDL TYPE	C TYPE
int	int32_t
long	int64_t
float	float
double	double
bool	typedef sidl_bool
char	char
string	char *
fcomplex	struct sidl_fcomplex
dcomplex	struct sidl_dcomplex
enum	enum
opaque	void *
interface	typedef
class	typedef
array	struct *

3. int32_t and int64_t for the SIDL int and long types;
4. a typedef for sidl_bool for the SIDL bool type;
5. preprocessor symbols TRUE and FALSE; and
6. function prototypes for the multi-dimensional array APIs for the basic SIDL types.

In general, clients don't need to worry about including sidl_header.h because the Babel generated header files will include it for you.

6.4 Mapping for classes, interfaces, arrays and r-arrays

Because C doesn't have built in mechanisms for protecting the global namespace, the C mapping attempts to avoid namespace collisions by using struct and method names that incorporate all the naming information from the package, class and method names. Without this approach, there would be multiple structures or functions with the same name which would cause compile and link errors. For a type Z in package X.Y, the name of the type that C clients use for an object reference is X_Y_Z. X_Y_Z is defined as follows in the X_Y_Z.h header file:

ANSI C
```
struct X_Y_Z__object;
struct X_Y_Z__array;
typedef struct X_Y_Z__object* X_Y_Z;
```

This code fragment also shows that struct X_Y_Z__array is used for a multi-dimensional array of X.Y.Z objects. Here are some additional concrete examples of the object and interface reference types derived by the C mapping:

ANSI C
```
/**
 * Symbol "sidl.BaseClass" (version 0.9.12)
 *
 * Every class implicitly inherits from <code>BaseClass</code>.  This
 * class implements the methods in <code>BaseInterface</code>.
 */
struct sidl_BaseClass__object;
struct sidl_BaseClass__array;
```

```
typedef struct sidl_BaseClass__object* sidl_BaseClass;

/**
 * Symbol "sidl.BaseInterface" (version 0.9.12)
 *
 * Every interface in <code>SIDL</code> implicitly inherits
 * from <code>BaseInterface</code>, and it is implemented
 * by <code>BaseClass</code> below.
 */
struct sidl_BaseInterface__object;
struct sidl_BaseInterface__array;
typedef struct sidl_BaseInterface__object* sidl_BaseInterface;
```

Here is an example of the C client-side binding for an r-array. This example is for the `solve` example from Section 5.4. Here, I assume that the package name is `num`, and the class name is `Linsol`. The data for each array is passed as a `double` pointer, and the index parameters are normal `in` ints.

ANSI C

```
/** C client-side API for solve method */
void num_Linsol_solve(/* in */ num_Linsol self,
                      /* in rarray[m,n] */ double* A,
                      /* inout rarray[n] */ double* x,
                      /* in */ int32_t m,
                      /* in */ int32_t n,
                      /* out */ sidl_BaseInterface *_ex);
```

The one catch for C programmers is that `A` is in column-major order — not the typical row-major ordering used in C. To access the element in row `i` and column `j`, you can use the `RarrayElem2(A,i,j,m)`. `RarrayElem2` is a convenience macro for C and C++ programmers to access r-arrays in column-major order, and it is defined in `sidlArray.h`. To access memory by stride one make `i`, the first index argument to `RarrayElem2`, your inner loop.

The additional argument `_ex` is used to indicate the presence of an exception when one has occured. In this case, the exception is possible because all methods implicitly throw *`sidl.RuntimeException`* . A non-`NULL` outcoming value in `*_ex` indicates that an exception occured. Exception handling is covered in Section 6.6.

Passing `NULL` for `A`, `x`, or `b` is not allowed. You must always pass a valid pointer.

6.5 Calling SIDL methods from C

The names of the C functions used to call SIDL methods are a concatenation of the package name, the class or interface name and the method name(s) with the period characters changed to underscores . If the method is specified as being overloaded (i.e., has a name extension) , the full method name is the concatenation of the package name, the class or interface name, the method name, *and* the type extension. For non-static methods, the object or interface pointer is passed as the first parameter before any of the formal parameters. This parameter operates like an `in` parameter. All class and interface methods have an extra `out` parameter at the end to hold an exception if one was thrown.

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 5.6. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

ANSI C

```
int32_t b1, i1, iresult, nresult;
sidl_BaseInterface ex;

Overload_Sample t  = Overload_Sample__create (&ex); SIDL_CHECK(ex);

nresult = Overload_Sample_getValue(t, &ex); SIDL_CHECK(ex);
```

```
  iresult = Overload_Sample_getValueInt(t, i1, &ex); SIDL_CHECK(ex);
  bresult = Overload_Sample_getValueBool(t, b1, &ex); SIDL_CHECK(ex);
```

SIDL_CHECK is used to check if an exception has been thrown. If an exception was thrown during the method call, it will goto the label EXIT. Exception handling is covered in more detail in Section 6.6.

Here are the C bindings for the critical addRef and deleteRef methods from sidl.BaseInterface. These methods are mentioned in particular because C clients must manage object reference counts themselves.

ANSI C

```
void
sidl_BaseInterface_addRef(/* in */ sidl_BaseInterface self,
                          /* out */ sidl_BaseInterface *_ex);

void
sidl_BaseInterface_deleteRef(/* in */ sidl_BaseInterface self,
                             /* out */ sidl_BaseInterface *_ex);
```

These same methods can be called from the sidl.BaseClass bindings. In fact, every C binding for an interface or class will have entries for addRef and deleteRef.

ANSI C

```
void
sidl_BaseClass_addRef(/* in */  sidl_BaseClass self,
                      /* out */ sidl_BaseInterface *_ex);

void
sidl_BaseClass_deleteRef(/* in */  sidl_BaseClass self,
                         /* out */ sidl_BaseInterface *_ex);
```

6.6 Catching and Throwing Exceptions in C

Every method can potentially throw an exception because all methods implicitly can throw *sidl.RuntimeException* . There is an extra out argument in the generated code that holds the exception. For maximum backward compatibility and consistency, the extra argument is of type sidl.BaseInterface. When the exception parameter is not NULL, it indicates that an exception has been thrown. When an exception is thrown, the caller should ignore the value of the other out parameters as well as the function's return value. Every time you call a method that potentially can throw an exception, you should check the result. Otherwise, those exceptions will be utterly ignored and potentially leak memory. You can write your custom code to check for exceptions and respond accordingly. Otherwise, there are are four macros provided in sidl_Exception.h to help with exception checking. Their use is fairly obvious from their names. They are:

ANSI C

```
SIDL_THROW(EX_VAR,EX_CLS,MSG)
SIDL_CHECK(EX_VAR)
SIDL_CLEAR(EX_VAR)
SIDL_CATCH(EX_VAR,sidl_NAME)
```

In these macros, EX_VAR is the exception object itself, EX_CLS is the name of the SIDL type we wish the exception to be in a string, MSG is the message we wish to include with the exception and a string, and sidl_NAME is the type of the exception we expect to catch, as a string.

The following SIDL method taken from the Babel regression tests demonstrates how exceptions are handled.

SIDL

```
int getFib(in int n, in int max_depth, in int max_value, in int depth)
  throws NegativeValueException, FibException;
```

Here is the C binding for this method:

```
int32_t
ExceptionTest_Fib_getFib(
  ExceptionTest_Fib self,
  int32_t n,
  int32_t max_depth,
  int32_t max_value,
  int32_t depth,
  sidl_BaseInterface *_ex);
```
ANSI C

Here is an example of how to perform exception handling in C using a package of macros defined in `sidl_Exception.h`. Note that the macros assume the exception class that is being thrown and caught inherits from or implements
sidl.BaseException — something guaranteed by Babel.

```
#include "sidl_Exception.h"
/* ...numerous lines deleted... */
  int x;
  sidl_BaseInterface _ex = NULL;

  x = ExceptionTest_Fib_getFib(f, 10, 1, 100, 0, &_ex);
  if (SIDL_CATCH(_ex, "ExceptionTest.TooDeepException")) {
    traceback(_ex);
    SIDL_CLEAR(_ex);
  }
  else if (SIDL_CATCH(_ex, "ExceptionTest.TooBigException")) {
    traceback(_ex);
    SIDL_CLEAR(_ex);
  }
  else if (_ex == NULL) {
    return FALSE;
  }
  SIDL_CHECK(_ex);
  return TRUE;

  EXIT:;
    traceback(_ex);
    SIDL_CLEAR(_ex);
    return FALSE;
```
ANSI C

You do not have to use the macros provided in `sidl_Exception.h` if you do no want to. You can check `_ex` by checking if it is not `NULL` and then trying to cast it to the various potential exception types.

The following code snippet shows how to throw an exception in C using the macros from `sidl_Exception.h`. The first argument to `SIDL_THROW` is the exception output parameter, and the second argument is the type of exception being thrown. The third argument provides a textual description of the exception.

```
#include "sidl_Exception.h"
/* ...numerous lines deleted... */
int32_t
impl_ExceptionTest_Fib_getFib(
  ExceptionTest_Fib self, int32_t n, int32_t max_depth, int32_t max_value,
    int32_t depth, sidl_BaseInterface* _ex)
{
  /* DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib) */
  if (n < 0) {
    SIDL_THROW(*_ex,
```
ANSI C

```
                ExceptionTest_NegativeValueException,
                "called with negative n");
  }
  /* ...lines deleted... */
  EXIT:;
    /* SIDL_THROW macro will jump here. */
    /* Clean up code should be here. */
    return theValue;
  /* DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib) */
}
```

The code section labeled `EXIT` is where you should put clean up code. The caller will ignore all the values leaving your C function (i.e., `out` or `inout` parameters) because you have thrown an exception, so your code should delete any references you were planning to return to the caller. It's good practice to set all `inout` and `out` array, interface or class pointers to `NULL`. This makes things work out better for clients who forget to check if an exception occurred or willfully choose to ignore it.

6.7 Implicitly defined methods

The C binding for interfaces and classes includes two methods for perform type casts. The methods are named `_cast` and `_cast2`. The leading underscore prevents these built in methods from conflicting with a user method because user methods cannot begin with an underscore. These methods increases the reference count of the underlying object if the cast succeeds — in Babel releases prior to 0.11.0 these methods did not increment the reference count. Every object has these two methods, we will use `sidl.BaseClass` as an example. Here are the signatures for `_cast` and `_cast2` from `sidl.BaseClass`.

ANSI C

```
sidl_BaseClass
sidl_BaseClass__cast(void* obj, /* out */ sidl_BaseInterface *_ex);

void*
sidl_BaseClass__cast2(void* obj, const char* type,
                      /* out */ sidl_BaseInterface *_ex);
```

The `_cast` method attempts to cast a SIDL interface or object pointer to a pointer to `sidl.BaseClass`. The `_cast2` method attempts to cast a SIDL interface or object pointer to a pointer to an interface or object pointer of the type named `type`. In the case of `_cast2`, the client is responsible for casting the return value into the proper pointer type. Both methods are `NULL` safe. A `NULL` return value indicates that the cast failed or that `obj` was `NULL`.

Non-abstract classes have an additional implicit method called `_create` to create new instances of the class. Interfaces and abstract classes do not have this method because you cannot instantiate them. The `_create` method returns a new reference that the client must manage. Here is an example of its signature.

ANSI C

```
/**
 * Constructor function for the class.
 */
sidl_BaseClass
sidl_BaseClass__create(/* out */sidl_BaseInterface *_ex);
```

6.8 Invoking Babel to generate C bindings

To create C stubs (i.e. code to support C clients to a set of SIDL classes or interfaces), you should invoke Babel as follows [1]:

[1] For information on additional command line options, refer to Section 3.2.

```
% babel --exclude-external --client=C file.sidl
```

or more cryptically

```
% babel -E -cC file.sidl
```

This will create more files than you can shake a stick at; although, the `--exclude-external` flag avoids generating files for symbols referenced in `file.sidl`. The files ending in `_IOR.h` and `_IOR.c` are the Intermediate Object Representation. The files ending with `_Stub.c` are the C stubs — the interface between a C client and the IOR. The remaining header files have external C API that C clients may use.

To use the C stubs, you must compile the stub files whose file names end with `_Stub.c` and link them against the SIDL runtime library and a backend implementation.

6.9 Invoking Babel to generate C implementations

To implement a set of SIDL classes in C, you should invoke Babel as follows:

```
% babel --exclude-external --server=C file.sidl
```

or use the short form

```
% babel -E -sC file.sidl
```

This will create a Makefile frament called `babel.make`, several C headers and source files. To create a working C implementation, the only files that need to be hand-edited are the C "Impl" files (header and source files that end in `_Impl.h` or `_Impl.c`. Changes to these files should be made between code splicer pairs. Code splicing is a technique Babel uses to preserve hand-edited code between multiple invocations of Babel. This allows a developer to refine their SIDL file without ruining all their previous implementations. Code between splicer pairs will be retained by subsequent invocations of Babel; code outside splicer pairs is not.

Here is an example of a code splicer pair in C.

ANSI C

```
/* DO-NOT-DELETE splicer.begin(num.Linsol._includes) */
/* Insert-Code-Here {num.Linsol._includes} (includes and arbitrary code) */
/* DO-NOT-DELETE splicer.end(num.Linsol._includes) */
```

The following example shows the Babel generate implementation file for the `solve` example from Section 5.4. The r-array data is presented as `double` pointers, and the index variables are normal integers.

ANSI C

```
void
impl_num_Linsol_solve(/* in */ num_Linsol self,
                      /* in rarray[m,n] */ double* A,
                      /* inout rarray[n] */ double* x,
                      /* in */ int32_t m,
                      /* in */ int32_t n,
                      /* out */ sidl_BaseInterface *_ex)
{
  *_ex = 0;
  /* DO-NOT-DELETE splicer.begin(num.Linsol.solve) */
  /* Insert-Code-Here {num.Linsol.solve} (solve method) */
  /* DO-NOT-DELETE splicer.end(num.Linsol.solve) */
}
```

The data for the 2-D array `A` is in column-major order. Use of the `RarrayElem2` macro to access `A` is covered above in Section 6.4.

Chapter 7

C++ Binding

Contents

7.1 Introduction

Babel has had two C++ bindings in its pre 1.0 history. This chapter deals with the newer of the two as the older one will no longer be supported in 1.0. However, it is helpful to be familiar with the history to understand why things are transitioning in the runup to a 1.0 release.

The original C++ binding was available since Babel's first public release (0.5.0 in July 2001). However, over the years of use and increasing features in Babel, it became clear that there were several deficiencies with the original C++ binding. Many of these deficiencies couldn't couldn't be remedied with simple patches... the entire C++ binding needed to be rethought from the ground up. Because C++ is arguably the most heavily used langauge binding, work began on a second C++ binding called the Utah C++ binding or simply "UCxx" as thanks to Steve Parker at University of Utah who took the first crack at actually implementing the new binding. The UCxx backend was released in Babel 0.10.0 in January 2005.

Within months, it was decided that the UCxx binding would become the only C++ binding for Babel 1.0. So in 0.11.0 the original C++ binding was renamed DCxx (for deprecated C++). The UCxx binding was also tweeked so the generated files had distinct suffices from the DCxx files. The UCxx binding already has all of its symbols generated in the `ucxx` namespace to avoid conflict with DCxx symbols. When Babel 1.0 is released, the DCxx binding will be gone, as will the `ucxx` namespace. The UCxx binding itself will be renamed Cxx, and the transition will be complete.

To maximize code's resilience to the forthcoming changes, we recommend that users take advantage of the `SIDL_USE_UCXX`, `UCXX`, and `UCXX_LOCAL` preprocessor macros. They are defined appropriately for the 0.11 release and will be undefined for the Babel 1.0 release. `SIDL_USE_UCXX` is meant to be used in `#ifdef SIDL_USE_CXX`/`#endif` blocks. If `SIDL_USE_UCXX` is defined, it means that the UCxx symbols are all in the `ucxx` namespace. Use `UCXX` where you would normally have `::ucxx` in your code, and use `UCXX_LOCAL` where you would normally have `ucxx::` in your code.

Table 7.1: SIDL to C++ Type Mappings

SIDL TYPE	C++ TYPE
int	int32_t
long	int64_t
float	float
double	double
bool	bool
char	char
string	std::string
fcomplex	sidl::fcomplex
dcomplex	sidl::dcomplex
enum	enum
opaque	sidl::opaque
interface	class
class	class
array	sidl::array (template specialization)

7.2 Basic Types

The basic types in SIDL are mapped into C++ according to Table 7.1.

7.3 SIDL C++ Header Suffix

The first thing that C++ users will notice is that C++ headers have a ".hh" suffix or a ".hxx" suffix to distinguish them from C's ".h" suffix. Pre Babel 0.11, the ".hh" suffix was used for all C++ bindings. For Babel 0.11 and after, the ".hh" suffix was exclusively for the DCxx binding, and ".hxx" was introduced for the UCxx binding.

7.4 SIDL's Main C++ Header File

All C++ code generated by Babel `#include`'s a file called "`sidl_ucxx.hh`". This file includes `babel_config.h`, the C header file that defines configuration information. Finally, `sidl_ucxx.hh` defines some C++ classes in the SIDL namespace such as

- `UCXX ::sidl::StubBase` [implementation detail] Common base class for all C++ stubs (proxy classes)
- template<T,U,V> UCXX ::sidl::basearray [implementation detail] Common base class for all C++ array classes.
- typedefs for `UCXX ::sidl::fcomplex`, `UCXX ::sidl::dcomplex`, and `UCXX ::sidl::opaque` (usually `std::complex`, `std::complex` and `void*`, respectively)
- `template<T> UCXX ::sidl::array` Template array type for SIDL arrays.
- template specializations [implementation detail] specialization of arrays of all SIDL types are defined in this file.

7.5 Calling Methods from C++

Since C++ is an object-oriented language, the language is much more amenable to the SIDL programming model and is less demanding of the programmer than bindings to non languages such as C or FORTRAN 77.

Table 7.2: SIDL Features Mapped onto C++

SIDL Feature	C++ Implementation
packages	C++ namespaces (no name transformations, but UCxx binding nests everything in a `ucxx` namespace until Babel 1.0 release.)
version numbers	ignored
interface	C++ class (called "stub", serves as a proxy to the implementation)
class	C++ class (called "stub", serves as a proxy to the implementation)
methods	C++ member functions; uses base method name when overloading; no name mangling; NOTE: Member functions beginning with a leading underscore(_) indicate a builtin function that is implicit by Babel or the C++ binding.
static methods	Static C++ member functions; uses base method name when overloading; no name mangling; even works for dynamically loaded object's exceptions thrown and caught using C++ exception handling.
reference counting	SIDL C++ stubs can be treated as smart-pointers. Constructors, destructors, and operators are overloaded so that explicit calls to `addRef()` or `deleteRef()` are rarely needed.
casting	Upcasting is safely handled with simple assignment. Downcasting should be done with `sidl::babel_cast<>()`. User should never call `dynamic_cast<>()` on a SIDL object since Babel's runtime system needs to be involved in verifying legality of the downcast. Downcasts should be checked by a call to (`_is_nil()`, or `_not_nil()`).
instance creation	Use static member function "`_create`". The default constructor for a C++ stub creates the equivalent of a NULL pointer. Works only with non-abstract classes.

These proxy classes (we call "stubs") serve as the firewall between the application in C++ and Babel's internal workings. As one would expect, the proxy classes maintain minimal state so that, unlike C or FORTRAN 77, there is no special context argument added to non-static member functions.

Below are examples using standard classes. The first is an example of creating an object of the base class and its association to the base interface.

C++
```
#ifdef SIDL_USE_UCXX
using namespace ucxx;
#endif

sidl::BaseClass object = sidl::BaseClass::_create();
sidl::BaseInterface interface = object;
```

Here is an example call to the addSearchPath in the SIDL.Loader class:

C++
```
#ifdef SIDL_USE_UCXX
using namespace ucxx;
#endif

std::string s("/try/looking/here");
sidl::Loader::addSearchPath( s );
```

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 5.6. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

C++
```
#ifdef SIDL_USE_UCXX
using namespace ucxx;
#endif

bool b1, bresult;
int  i1, iresult, nresult;

Overload::Sample t  = Overload::Sample::_create();

nresult = t.getValue();
bresult = t.getValue(b1);
iresult = t.getValue(i1);
```

7.6 Catching and Throwing Exceptions in C++

Adapted from the Babel regression tests, the following is an example of a package called ExceptionTest that has a class named Fib with a method declared in SIDL as follows:

SIDL
```
int getFib(in int n, in int max_depth, in int max_value, in int depth)
  throws NegativeValueException, FibException;
```

The corresponding C++ code fragment to use this method is:

C++
```
#ifdef SIDL_USE_UCXX
using namespace ucxx;
#endif

ExceptionTest::Fib fib = ExceptionTest::Fib::_create();
```

```
try {
  int result = fib.getFib( 4, 100, 32000, 0 );
  cout << "Result of fib.getFib() = " << result << endl;
} catch ( ExceptionTest::NegativeValueException  e ) {
  // ...
} catch ( ExceptionTest::FibException e ) {
  // ...
}
```

This example shows the standard way to throw an exception in C++. You are not strictly required to call the `setNote` and `add` methods; however, these methods provide information that may be helpful in debugging or error reporting.

C++

```
int32_t
ExceptionTest::Fib_impl::getFib_impl (
  /*in*/ int32_t n,
  /*in*/ int32_t max_depth,
  /*in*/ int32_t max_value,
  /*in*/ int32_t depth )
throw (
  UCXX ::ExceptionTest::FibException,
  UCXX ::ExceptionTest::NegativeValueException,
  UCXX ::sidl::RuntimeException
){
  // DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib)
  if (n < 0) {
    UCXX ::ExceptionTest::NegativeValueException ex =
      UCXX ::ExceptionTest::NegativeValueException::_create();
    ex.setNote("n negative");
    ex.add(__FILE__, __LINE__, "ExceptionTest::Fib_impl::getFib");
    throw ex;

  } else if (depth > max_depth) {
    UCXX ::ExceptionTest::TooDeepException ex =
      UCXX ::ExceptionTest::TooDeepException::_create();
    ex.setNote("too deep");
    ex.add(__FILE__, __LINE__, "ExceptionTest::Fib_impl::getFib");
    throw ex;

  } else if (n == 0) {
    return 1;

  } else if (n == 1) {
    return 1;

  } else {
    int32_t a = getFib(n-1, max_depth, max_value, depth+1);
    int32_t b = getFib(n-2, max_depth, max_value, depth+1);
    if (a + b > max_value) {
      UCXX ::ExceptionTest::TooBigException ex =
        UCXX ::ExceptionTest::TooBigException::_create();
      ex.setNote("too big");
      ex.add(__FILE__, __LINE__, "ExceptionTest::Fib_impl::getFib");
      throw ex;
    }
```

```
    return a + b;
  }
  // DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib)
}
```

7.7 Invoking Babel to generate C++ stubs

To create the C++ stubs from a SIDL file, invoke Babel as follows [1]:

```
% babel --exclude-external --client=UC++ file.sidl
```

or simply

```
% babel -E -cUC++ file.sidl
```

This will create a babel.make file, some C headers and sources, and many C++ headers and sources. Files ending in ".c" or ".h" are in C, files ending in ".cxx" or ".hxx" are the UC++ binding. You will need to compile and link the files together to use the C++ stubs.

7.8 Implementing SIDL Classes in C++

Much of the information from the previous section is pertinent to implementing a SIDL class in C++. The types of the arguments are as indicated in Table 7.1. Your implementation can call other SIDL methods, in which case follow the rules for client calls.

To create the implementation, you must first have a valid SIDL file, then invoke Babel as follows:

```
% babel --exclude-external --server=UC++ file.sidl
```

or simply

```
% babel -E -sUC++ file.sidl
```

This will create a makefile fragment called babel.make, several C headers and source files, and numerous C++ header and source files. To create a working implementation, the only files that need to be hand-edited are the C++ "Impl" files (header and source files that end in `_Impl.hxx` or `_Impl.cxx`). All your additions to this file should be made between code splicer pairs. Code splicing is a technique Babel uses to preserve hand-edited code between multiple invocations of Babel. This allows a developer to refine their SIDL file without ruining all their previous implementations. Code between splicer pairs will be retained by subsequent invocations of Babel; code outside splicer pairs is not.

Here is an example of a code splicer pair in C++. In this example, you would replace the line "// Insert code here... " with your implementation.

C++

```
void MyPackage::MyClass_impl::myMethod_impl() {
    // DO-NOT-DELETE splicer.begin(MyPackage.MyClass.myMethod)
    // Insert code here...
    // DO-NOT-DELETE splicer.end(MyPackage.MyClass.myMethod)
}
```

[1] For information on additional command line options, refer to Section 3.2.

It is important to understand where and why splicer blocks occur. Splicer blocks appear at the beginning and end of each Impl header and source file; for developers to add `#include`'s and other miscellaneous items respectively. In the headers, there is a splicer block that allows a user to make the impl class inherit from some other class. From SIDL's point of view this is private inheritance — meaning that it is useful for inheriting implementation details, but they can't be automatically exposed to the SIDL method dispatch mechanism. There is a splicer block inside the class definition for developers to add any data members the wish to the class. In the source files, splicer blocks appear in each method implementation. There are two implicit methods (i.e., methods that did not appear in the SIDL file) that must also be implemented. The _ctor method is a constructor function that is run whenever an object is created. The _dtor method is a destructor function that is run whenever an object is destroyed. If the object has no state, these functions are typically empty.

7.9 Accessing SIDL Arrays From C++

Although it is feasible to expose the underlying C array API to create, destroy and access array elements and meta-data, the C++ bindings provide a `sidl::array<T>` template mechanism that is more in keeping with C++ idioms.

For SIDL built-in types, template specializations of `sidl::array<T>` are defined in `sidl_ucxx.hxx`. For SIDL interface and classes, the array template is again specialized in the corresponding stub header. The reason for the extensive use of template specialization is an effort to hide the details that the array implementation shifts between the C++ type externally, and the C-based types stored in the IOR. (See `basearray` in `sidl_ucxx.hxx` for the traits classes and grungy implementation details.)

An example is given below.

C++
```
int32_t len = 10; // array length=10
int32_t dim = 1;  // one dimensional
int32_t lower[1] = {0}; // zero offset
int32_t upper[1] = {len-1};
int32_t prime = nextPrime(0);

// create a SIDL array of primes.
sidl::array<int32_t> a = sidl::array<int32_t>::createRow(dim, lower, upper);
for( int i=0; i<len; ++i ) {
   prime = nextPrime( prime );
   a.set(i, v);
}
```

Of course, the example above is only one way to create an array. The list of member functions for all C++ array classes is:

C++
```
// constructors
array ( ior_array_t * src ); // internal
array ( const array & src ); // copy constructor

// destructor
~array() ;

// create row-size of 1 to 7 dimensions
static array<T>
createRow( int32_t dimen, const int32_t lower[],
           const int32_t upper[]);

// create column-wise of 1 to 7 dimensions
static array<T>
createCol( int32_t dimen, const int32_t lower[],
           const int32_t upper[]);
```

```
// create 1-D array of specified length
static array<T> create1d( int32_t len );

// create 1-D array of specified length and init
static array<T> create1d(int32_t len, ior_item_internal_t data)

// create 2-D array of specified extents
static array<T> create2dCol( int32_t m, int32_t n);

// create 2-D array of specified extents
static array<T> create2dRow( int32_t m, int32_t n);

// get a slice of the array
array<T>
slice( int32_t dimen, const int32_t numElem[],
       const int32_t *srcStart = 0,
       const int32_t *srcStride = 0,
       const int32_t *newStart = 0);

void borrow( item_ior_t * first_element, int32_t dimen,
        const int32_t lower[], const int32_t upper[],
        const int32_t stride[]);

void ensure( int32_t dimen, array_ordering ordering );

void addRef();

void deleteRef();

// get/set
cxx_item_t get(int32_t i);
cxx_item_t get(int32_t i1, int32_t i2);
cxx_item_t get(int32_t i1, int32_t i2, int32_t i3);
cxx_item_t get(int32_t i1, int32_t i2, int32_t i3,
                       int32_t i4);
cxx_item_t get(int32_t i1, int32_t i2, int32_t i3,
                       int32_t i4, int32_t i5);
cxx_item_t get(int32_t i1, int32_t i2, int32_t i3,
                       int32_t i4, int32_t i5, int32_t i6);
cxx_item_t get(int32_t i1, int32_t i2, int32_t i3,
                       int32_t i4, int32_t i5, int32_t i6, int32_t i7);
cxx_item_t get(const int32_t *indices);

void set(int32_t i, cxx_item_t elem);
void set(int32_t i1, int32_t i2, cxx_item_t elem);
void set(int32_t i1, int32_t i2, int32_t i3,
         cxx_item_t elem);
void set(int32_t i1, int32_t i2, int32_t i3, int32_t i4,
         cxx_item_t elem);
void set(int32_t i1, int32_t i2, int32_t i3, int32_t i4,
         int32_t i5, cxx_item_t elem);
void set(int32_t i1, int32_t i2, int32_t i3, int32_t i4,
         int32_t i5, int32_t i6, cxx_item_t elem);
```

```
void set(int32_t i1, int32_t i2, int32_t i3, int32_t i4,
         int32_t i5, int32_t i6, int32_t i7, cxx_item_t elem);
void set(const int32_t *indices, cxx_item_t elem);

// [] overloaded to be same as get(i)
cxx_item_t operator[](int32_t i) const ;

bool is1dPacked() const;

// returns STL forward iterator iff 1DPacked, else null
iterator begin();

// returns STL forward iterator iff 1DPacked, else null
const_iterator begin();

// returns STL forward iterator iff 1DPacked, else null
iterator end();

// returns STL forward iterator iff 1DPacked, else null
const_iterator end();

const int32_t* first() const;

int32_t* first();

void copy( const array< T >& src );


// other accessors
int32_t dimen() const;

int32_t lower( int32_t dim ) const;

int32_t upper( int32_t dim ) const;

int32_t stride( int32_t dim ) const;

bool _is_nil() const;

bool _not_nil() const;

// get a const pointer to the actual array ior
const array_ior_t* _get_ior() const;

// get a non-const pointer to the actual array ior
array_ior_t* _get_ior();

void _set_ior( ior_array_t * s);

array& operator =(const array &rhs);

array& operator =(const basearray &rhs);
```

where `cxx_array_t`, `cxx_item_t`, `ior_array_t`, `ior_item_t`, `ior_item_internal_t`, `iterator`, `const_iterator`,

pointer, and value_type are all public typedefs in the array class. The values of these typedefs are determined by traits classes. Traits classes are a fairly standard, albeit advanced, C++ templating idiom. Refer to any advanced C++ text for detailed explanation. As an example, we reproduce both the array_traits<> and array<> template specializations for int32_t here to show how what these typedefs are for that special case. Refer to sidl_ucxx.hxx for more built-in types and the UCxx stubs for the user-defined types.

C++
```
// template specialization for array_traits<int32_t>
 template<>
 struct array_traits<int32_t> {
   typedef array<int32_t>                              cxx_array_t;
   typedef int32_t                                  cxx_item_t;
   typedef struct sidl_int__array                          ior_array_t;
   typedef int32_t                                  ior_item_t;
   typedef const int32_t*      ior_item_internal_t;
   typedef cxx_item_t                                  value_type;
   typedef value_type*                                 pointer;
   typedef const value_type*                           const_pointer;
 };
template<>
 class array< int32_t >
   : public basearray
 {
 public:
   typedef basearray                                     Base;
   typedef array_traits<int32_t>::cxx_array_t             cxx_array_t;
   typedef array_traits<int32_t>::cxx_item_t              cxx_item_t;
   typedef array_traits<int32_t>::ior_array_t             ior_array_t;
   typedef array_traits<int32_t>::ior_item_t              ior_item_t;
   typedef array_traits<int32_t>::ior_item_internal_t     ior_item_internal_t;
   typedef array_iter< array_traits<int32_t> >            iterator;
   typedef const_array_iter< array_traits<int32_t> >      const_iterator;
   typedef array_traits< int32_t > ::pointer           pointer;
   typedef array_traits< int32_t > ::value_type        value_type;

   // lots of methods to follow
 }
```

The C++ mapping for r-arrays is essentially identical to the mapping for C (see Section 6.4). The only difference is that the C++ client header provides an overloaded version of each method containing an r-array taking normal SIDL arrays instead of raw data. For example, the solve method from Section 5.4 produces the following code in the client-side header file.

C++
```
   void solve (/*in*/    double* A,
               /*inout*/ double* x,
               /*in*/    double* b,
               /*in*/    int32_t m,
               /*in*/    int32_t n) throw ();

   void solve (/*in*/    ::sidl::array<double> A,
               /*inout*/ ::sidl::array<double>& x,
               /*in*/    ::sidl::array<double> b) throw();
```

Please note that multi-dimensional arrays, such as A in this case, are stored in column-major ordering. Babel provides macros to access r-array data correctly. In this example, you can use RarrayElem2(A, i, j, m) to access the element in row i and column j. There are similar macros for arrays of dimension 1 through 7 in sidlArray.h.

If you were implementing `solve` in C++, the Babel generated implementation file for it would look like this:

C++

```
void num::Linsol_impl::solve (/*in*/    double* A,
                              /*inout*/ double* x,
                              /*in*/    double* b,
                              /*in*/    int32_t m,
                              /*in*/    int32_t n ) throw ()
{
  // DO-NOT-DELETE splicer.begin(num.Linsol.solve)
  // insert implementation here
  // DO-NOT-DELETE splicer.end(num.Linsol.solve)
}
```

To access memory by stride one make the row index your inner loop and the column index your outer loop.

7.10 C++ Specific Babel Command Line Options

The Babel C++ binding has one command line option particular to it. Using the option –cxx-ior-exception (or it's short form '-x') will generate C++ Babel stubs that check for a null IOR whenever a method is called on them. If a method is called on a Stub holding a null IOR, it will throw a NullIORException. If this option is not passed to Babel, the program will simply crash, as C++ would do normally with a null pointer.

Chapter 8

FORTRAN 77 Bindings

Contents

8.1 Introduction

This chapter provides an introduction to Babel's FORTRAN 77 bindings. Babel supports both callers and callees written in FORTRAN 77 so this chapter illustrates the use of Babel for both. That is, it shows how to use Babel to wrap the implementation of software written in FORTRAN 77 as well as how to call software, possibly implemented in any other supported language, from FORTRAN 77.

8.2 Basic Types

For pointer types, such as opaque, interface, class, and array, a 64-bit integer is used, so FORTRAN 77 code will be portable between systems with a 32 bit address space and systems with a 64 bit address space. On a 32 bit system, the upper 32 bits of these quantities are ignored. Systems with more than 64-bit pointers aren't currently supported.

Generally, clients should treat opaque, interface, class and array values as black boxes. However, there is one value that is special. A value of zero for any of these quantities indicates that the value does not refer to an object. Zero is the FORTRAN 77 equivalent of NULL . Any nonzero value is or should be a valid object reference. Developers writing in FORTRAN 77 should initialize values to be passed as `in` or `inout` parameters to zero or a valid object reference.

When mapping the SIDL string type into FORTRAN 77, some capability was sacrificed to make it possible to use normal looking FORTRAN 77 string handling. One difference is that all FORTRAN 77 strings have a limited fixed size. When implementing a subroutine with an `out` parameter, the size of the string is limited to 512 characters. This can be changed when configuring babel by changing the value of `SIDL_F77_STR_MINSIZE` in `runtime/sidl/babel_config.h` before compiling and installing babel.

Enumerated types are just integer values. The constants are defined in an includable file assuming your FORTRAN 77 compiler supports some form of including.

Table 8.1: SIDL to FORTRAN 77 type mapping

SIDL TYPE	FORTRAN 77 TYPE
int	INTEGER*4
long	INTEGER*8
float	REAL
double	DOUBLE PRECISION
bool	LOGICAL
char	CHARACTER*1
string	CHARACTER*(*)
fcomplex	COMPLEX
dcomplex	DOUBLE COMPLEX
enum	INTEGER
opaque	INTEGER*8
interface	INTEGER*8
class	INTEGER*8
array	INTEGER*8

8.3 Calling Methods From FORTRAN 77

All SIDL methods are implemented as FORTRAN 77 subroutines regardless of whether they have a return value or not. For object methods (as opposed to static methods), the object or interface pointer is passed as the first argument to the subroutine before all the formally declared arguments. The exception is static methods, where the object or interface pointer does not appear in the argument list at all.

When a method has a return value, a variable to hold the return value should be passed as an argument following the formally declared arguments. This extra argument behaves like an *out* parameter.

All methods can potentially throw exceptions, and the exception pointer is passed as an extra argument after the return value argument (if any). The exception argument behaves like an *out* parameter. A non-zero value indicates that an exception has been thrown. Exception handling for FORTRAN 77 is covered in greater depth in Section 8.4.

The name of the subroutine that FORTRAN 77 clients should call is derived from the fully qualified name of the class and the name(s) of the method. If the method is specified as overloaded (i.e., has a name extension), the method's full name will be used. That is, the concatenation of the short name and the name extension will be used for a unique method name. Hence, to determine the subroutine name for FORTRAN 77, take the fully qualified name, replace all the periods with underscores, append an underscore, append the short method name, append the method name extension (if any) and then append "_f".

These general rules will be illustrated with a set of practical examples. For the time being, the example codes will not check the *out* except argument that indicates if an exception has occurred. For example, to call the deleteRef() method on a *sidl.BaseInterface* interface, you would write:

Fortran 77

```
      integer*8 interface1, except
C     code to initialize interface1 here
      call sidl_BaseInterface_deleteRef_f(interface1, except)
```

To call the isSame method on a *sidl.BaseInterface*, you would write:

Fortran 77

```
      integer*8 interface1, interface2, except
      logical areSame
C     code to initialize interface1 and interface2 here
      call sidl_BaseInterface_isSame_f(interface1,
     $     interface2, areSame, except)
C     now areSame holds the return value
```

To call the isType method on a *sidl.BaseInterface*, you would write:

```
      integer*8 interface1, except                                    Fortran 77
      logical typeMatch
C     code to initialize interface1 here
      call sidl_BaseInterface_isType_f(interface1, 'x.y.z',
     $   typeMatch, except)
```

Here is a code snippet to find the name of the SIDL class that implements a particular *sidl.BaseInterface* object. This example requires a sequence of related calls to SIDL methods.

```
      integer*8 interface1, classinfo, except                         Fortran 77
      character*256 className
C     code to initialize interface1 here
      call sidl_BaseInterface_getClassInfo_f(interface1,
     $   classinfo, except)
      call sidl_ClassInfo_getName(classinfo, className, except)
      call sidl_BaseInterface_deleteRef_f(classinfo, except)
```

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 5.6. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

```
      integer*8 t, except                                             Fortran 77
      logical b1, bretval
      integer*4 i1, iretval

      call Overload_Sample__create_f (t, except)

      call Overload_Sample_getValue_f (t, iretval, except)
      call Overload_Sample_getValueInt_f (t, i1, iretval, except)
      call Overload_Sample_getValueBool_f (t, b1, bretval, except)
```

For interfaces and classes, there are two implicit methods called `_cast()` and `_cast2()`. Both of these methods are used to convert from one type to another, and each can be used for upcasting up downcasting. If the cast succeeds, these method calls increment the reference count, and the caller owns the returned reference. In Babel releases prior to 0.11.0, these methods were special cases that did not increment the reference count, but now they behave like normal methods. The returned reference is owned by the caller.

`_cast()` is a static method. It tries to convert its opaque argument to the type of the class indicated by the method name. For example, `x_y_z__cast(obj, xyz, except)` will try to convert `obj` to type *x.y.z*. If `except` is nonzero, an exception occurred; otherwise, if `xyz` is nonzero, the cast was successful.

`_cast2()` is an object method. Its return type is opaque, and it has one formal argument, a string in addition to the implicit object/interface reference. The `_cast()` method attempts to cast the object/interface to the named type.

For non-abstract classes, there is an implicit method called `_create()`. It creates and returns an instance of the class.

Here are examples of the use of these two methods:

```
      integer*8 object, interface, except                             Fortran 77
      call sidl_BaseClass__create_f(object, except)
      call sidl_BaseInterface__cast_f(object, interface, except)
c     the following call to _cast2 is equivalent to the previous _cast call
      call sidl_BaseClass__cast2_f(object, 'SIDL.BaseInterface',
     $     interface, except)
```

Please note the presence of two underscores between BaseClass and create and between BaseClass and cast; the extra underscore is there because the first character of the method name is an underscore.

Below is an example call to the `addSearchPath()` in the *sidl.Loader* class. This is an example of calling a static method.

```
      integer*8 except                                                    Fortran 77
      call sidl_Loader_addSearchPath_f('/try/looking/here', except)
```

Your FORTRAN 77 must manage any object references created by the calls you make. This means you must call either `deleteRef` on any object references returned by a subroutine call or give the object reference to another part of the code that will take responsibility for `deleteRef`'ing it. For example, you can return an object reference to your caller, and they will assume responsibility for the object reference.

8.4 Catching and Throwing Exceptions in FORTRAN 77

All SIDL methods can throw an exception; hence, an extra variable of type `INTEGER*8` must be passed to hold a pointer if an exception is thrown. All methods implicitly throw *sidl.RuntimeException* even if the SIDL declaration does not have a *throws* clause. The *sidl.RuntimeException* is used primarily to indicate that an error occurred in the Babel generated code or that a network failure occurred for distributed Babel calls.

For maximum backward compatibility, the base exception type argument is *sidl.BaseInterface* though the base exception class is *sidl.SIDLException*. The exception argument appears after the return value when both occur in a method. After the call, the client should test this argument. If a function does not test the exception argument, thrown exceptions will be utterly ignored — not propagated to higher level functions. If the exception parameter is non-zero, an exception was thrown by the method, and the method should respond appropriately. When an exception is thrown, the value of all other arguments is undefined.

When the exception parameter is non-zero, your code should try casting it to each of the possible exceptions in turn. A successful cast indicates the type of exception that has occured. If one of the possible exception types is a subclass of another one, you should try casting to the subclass before casting to the super class — assuming that the distinction between the two exception types results in different exception recovery behavior.

sidl.BaseException defines two methods that can be helpful when reporting exceptions to end users: `getNote` and `getTrace`. `getNote` often provides some indication of what went wrong. Its contents are provided by the implementor of the function you called, so it can be empty. Similarly, `getTrace` make provide a summary of the call stack. Again, it depends on implementors to provide information.

One approach to exception handling is to pass the exception on to your caller. In this case, you should call *sidl.BaseException.add* to add another line in the stack trace for the exception.

Here is another example adapted from the Babel regression tests. Package ExceptionTest has a class named Fib with a method declared in SIDL as follows:

```
int getFib(in int n, in int max_depth, in int max_value, in int depth)   SIDL
    throws NegativeValueException, FibException;
```

Here is the outline of a FORTRAN 77 code fragment to use this method. When an exception is thrown, the value of the *out* and *inout* parameters is unknown, the best practice is to ignore their values.

```
      integer*8 fib, except, except2, except3                             Fortran 77
      integer*4 index, maxdepth, maxval, depth, result
      call ExceptionTest_Fib__create_f(fib, except)
      if (except .ne. 0) thn
C        do something with a runtime exception
      endif
      index = 4
      maxdepth = 100
      maxvalue = 32000
      depth = 0
      call ExceptionTest_getFib_f(fib, index, maxdepth,
     $       maxvalue, depth, result, except)
      if (except .ne. 0) then
        call ExceptionTest_FibException__cast_f(except, except2, except3)
        if (except2 .ne. 0) then
c          do something here with the FibException
```

```
            call ExceptionTest_FibException_deleteRef_f(except2, except3)
          else
            call ExceptionTest_NegativeValueException__cast_f
     $           (exception, except2, except3)
c           do something here with the NegativeValueException
            call ExceptionTest_NegativeValueException_
     $           deleteRef_f(except2, except3)
          endif
          call sidl_BaseException_deleteRef_f(except, except3)
        else
          write (*,*) 'getFib for ', index, ' returned ', result
        endif
        call ExceptionTest_Fib_deleteRef_f(fib, except)
```

Here is an example of FORTRAN 77 code that throws an exception. Note that not all exceptions are checked in this example. In general, it is safe to assume that calling `deleteRef`, `_cast` or `_cast2` on an exception will never cause an exception to be thrown because returned exceptions are always local.

Fortran 77

```
        subroutine ExceptionTest_Fib_getFib_fi(self, n, max_depth,
     &     max_value, depth, retval, exception)
        implicit none
        integer*8 self, exception
        integer*4 n, max_depth, max_value, depth, retval
C       DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib)
        character*(*) myfilename
        parameter(myfilename='ExceptionTest_Fib_Impl.f')
C ...lines of code deleted...
        if (n .lt. 0) then
          call ExceptionTest_NegativeValueException__create_f(exception)
          if (exception .ne. 0) then
            call ExceptionTest_NegativeValueException_setNote_f(
     $             exception,
     $             'called with negative n')
            call ExceptionTest_NegativeValueException_add_f(
     $             exception,
     $             myfilename,
     $             57,
     $             'ExceptionTest_Fib_getFib_impl')
            return
          endif
C ...lines of code deleted...
C       DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib)
        end
```

Please note that when your code throws an exception it should `deleteRef` any references it was planning to return to its caller because the caller is instructed to ignore any values returned when an exception is thrown. Any caller of a method that returns an exception should ignore the values of *`out`* and *`inout`* parameters, so anything you do not free will become a reference and memory leak. In general, it is good practice to set all *`out`* and *`inout`* array, class and interface arguments to zero before returning when throwing an exception. This makes things work out better for clients who forget to check if an exception occurred or willfully choose to ignore it.

8.5 Invoking Babel to generate FORTRAN 77 Stubs

Here is how you should invoke Babel to create the FORTRAN 77 stubs for an IDL file [1].

```
% babel --exclude-external --client=f77 file.sidl
```

or simply

```
% babel -E -c=f77 file.sidl
```

This will create a babel.make file, numerous C headers, numerous C source files, and some FORTRAN 77 files. The files ending in `_fStub.c` are the FORTRAN 77 stubs that allow FORTRAN 77 to call a SIDL method. The files ending in `.fif` are documentation for FORTRAN 77 programmers. This shows you how the class and interface would have been defined if they were implemented in FORTRAN 77. The `.fif` files should **not** be compiled; they are only for reference.

You will need to compile and link the files ending in `_fStub.c` into your application (i.e. `STUBSRCS` in `babel.make`). Normally, the IOR files (`_IOR.c`) are linked together with the implementation file, so you normally do not need to compile them.

If you have some *enum*'s defined in your SIDL file, Babel will generate FORTRAN 77 include files in the style of DEC FORTRAN (Compaq FORTRAN? (now HP Fortran???)) `%INCLUDE`. These files are named by taking the fully qualified name of the *enum*, changing the periods to underscores, and appending `.inc` . Here is an example of a generated include file.

Fortran 77

```
C       File:           enums_car.inc
C       Symbol:         enums.car-v1.0
C       Symbol Type:    enumeration
C       Babel Version:  0.5.0
C       Description:    Automatically generated; changes will be lost
C
C       babel-version = 0.5.0
C       source-line   = 25
C
        integer porsche
        parameter (porsche = 911)
        integer ford
        parameter (ford = 150)
        integer mercedes
        parameter (mercedes = 550)
```

8.6 Implementing Classes in FORTRAN 77

Much of the information from the previous section is pertinent to implementing a SIDL class in FORTRAN 77. The types of the arguments are as indicated in Table 8.1. Your implementation can call other SIDL methods in which case follow the rules for client calls.

You should invoke Babel:

```
% babel --exclude-external --server=f77 file.sidl
```

or simply

[1] For information on additional command line options, refer to Section 3.2.

```
% babel -E -s=f77 file.sidl
```

This will create a babel.make, numerous C headers, numerous C source files and some FORTRAN 77 source files. Your job is to fill in the FORTRAN 77 source files with the implementation of the methods. The files you need to edit all end with `_Impl.f`. All your changes to the file should be made between code splicer pairs. Code between splicer pairs will be retained by subsequent invocations of Babel; code outside splicer pairs is not. Here is an example of a code splicer pair. In this example, you would replace the line "C Insert-Code-Here..." with your lines of code.

Fortran 77

```
C       DO-NOT-DELETE splicer.begin(_miscellaneous_code_start)
C       Insert-Code-Here {_miscellaneous_code_start} (extra code)
C       DO-NOT-DELETE splicer.end(_miscellaneous_code_start)
```

Each _Impl.f file contains numerous empty subroutines. Each subroutine that you must implement is partially implemented. The `SUBROUTINE` statement is written, and the types of the arguments have been declared. You must provide the body of each subroutine that implements the expected behavior of the method.

There are two implicit methods (i.e. methods that did not appear in the SIDL file) that must also be implemented. The `_ctor()` method is a constructor function that is run whenever an object is created. It's purpose is to initialize the object to make it ready for any of the other method calls. The `_dtor()` method is a destructor function that is run whenever an object is destroyed. The destructor's purpose is to free any resources allocated by the object. If the object has no state, these functions are typically empty.

The SIDL IOR keeps a pointer (i.e., a C `void *`) for each object that is intended to hold a pointer to the object's internal data. The FORTRAN 77 skeleton provides two functions that the FORTRAN 77 developer will need to use to access the private pointer. The name of the function is derived from the fully qualified type name as follows. Replace periods with underscores and append `__get_data_f` or `__set_data_f`. The first argument is the object pointer (i.e. self), and the second argument is an opaque . These arguments are 64 bit integers in FORTRAN 77, but the number of bits actually stored by the IOR is determined by the `sizeof(void *)`.

Babel/SIDL does not provide a low level mechanism for FORTRAN 77 to allocate memory to use for the private data pointer; however, there is an example of a FORTRAN 77 object with state in Section 8.8.

8.7 Accessing SIDL Arrays From FORTRAN 77

For FORTRAN 77, the difference in how you access normal SIDL arrays and r-arrays is profound. Normal SIDL arrays are passed in as an `integer*8`, and you either access them using an function API or by converting the array data to a index into a known array. R-arrays appear like normal FORTRAN 77 arrays, so there is a big incentive to use r-arrays unless you cannot.

The client-side interface for the `solve` example introduced in Section 5.4 behaves as if it is a FORTRAN 77 function with the following declarations:

Fortran 77

```
      subroutine num_Linsol_solve_f(self, A, x, m, n, exception)
      implicit none
C     in num.Linsol self
      integer*8 self
C     in int m, n
      integer*4 m, n
C     out sidl.BaseInterface exception
      integer*8 exception
C     in rarray<double,2> A(m,n)
      double precision A(0:m-1, 0:n-1)
C     inout rarray<double> x(n)
      double precision x(0:n-1)
      end
```

FORTRAN 77 programmers should note that the array indices go from **0** to `m`−1 instead of the normal 1 to `m`. This is a concession to the C and C++ programmers who have to deal with the fact that A is stored in column-major order.

On the server-side, the interface for `solve` appears as follows:

Fortran 77

```
      subroutine num_Linsol_solve_fi(self, A, x, m, n, exception)
      implicit none
C     in num.Linsol self
      integer*8 self
C     in int m
      integer*4 m
C     in int n
      integer*4 n
C     out sidl.BaseInterface exception
      integer*8 exception
C     in rarray<double,2> A(m,n)
      double precision A(0:m-1, 0:n-1)
C     inout rarray<double> x(n)
      double precision x(0:n-1)

C     DO-NOT-DELETE splicer.begin(num.Linsol.solve)
C     Insert-Code-Here {num.Linsol.solve} (solve method)
C     DO-NOT-DELETE splicer.end(num.Linsol.solve)
      end
```

Note again that the array indices go from 0 to m−1. The implementation should avoid changing the data in *in* parameters.

The remainder of this section is dedicated to how you access normal SIDL arrays. The normal SIDL C function API is available from FORTRAN 77 to create, destroy and access array elements and meta-data. The function name for FORTRAN has _f appended. Note that array functions do not throw exceptions.

For SIDL types *dcomplex*, *double*, *fcomplex* , *float*, *int* and *long*, SIDL provides a method to get direct access to the array elements. For the other types, you must use the functional API to access array elements.

For type X, there is a FORTRAN 77 function called sidl_X__array_access_f to provide a method to get direct access. An example is given below. Of course, this will not work if your FORTRAN 77 compiler does array bounds checking.

Fortran 77

```
      integer*4 lower(1), upper(1), stride(1), i, index(1)
      integer*4 value, refarray(1), modval
      integer*8 nextprime, refindex, tmp
      lower(1) = 0
      value = 0
      upper(1) = len - 1
      call sidl_int__array_create_f(1, lower, upper, retval)
      call sidl_int__array_access_f(retval, refarray, lower,
     $     upper, stride, refindex)
      do i = 0, len - 1
         tmp = value
         value = nextprime(tmp)
         modval = mod(i, 3)
         if (modval .eq. 0) then
            call sidl_int__array_set1_f(retval, i, value)
         else
            if (modval .eq. 1) then
               index(1) = i
               call sidl_int__array_set_f(retval, index, value)
            else
C this is equivalent to the sidl_int__array_set_f(retval, index, value)
               refarray(refindex + stride(1)*(i - lower(1))) =
     $              value
            endif
```

```
          endif
        enddo
```

To access a two dimensional array, the expression referring to element `i`, `j` is

```
      refarray(refindex + stride(1) * (i - lower(1)) + stride(2) * (j - lower(2))
```
Fortran 77

To access a three dimensional array, the expression referring to element `i`, `j`, `k` is

```
      refarray(refindex + stride(1) * (i - lower(1)) + stride(2) *
     $   (j - lower(2)) + stride(3) * (k - lower(3))
```
Fortran 77

You can call things like LINPACK or BLAS if you want, but you should check the stride to make sure the array is packed as you need it to be. `stride(i)` indicates the distance between elements in dimension `i`. A value of 1 means elements are packed densely in dimension `i`. Negative stride values are possible, and when an array is a slice of another array, there may be no dimension with a stride of 1.

For a *dcomplex* array, the reference array should a FORTRAN array of `REAL*8` instead of a FORTRAN array of double complex to avoid potential alignment problems. For a *fcomplex* array, the reference array is a `COMPLEX*8` because we don't anticipate an alignment problem in this case.

8.8 FORTRAN 77 objects with state

If you need to implement a FORTRAN 77 class with state, you can use SIDL arrays to store the state information. This is certainly not the only way to implement a FORTRAN 77 class with state, but it's one that will work wherever Babel works. For example, if you have a class whose state requires three boolean variables and two double precision variables, your constructor might look something like the following:

Fortran 77
```
        subroutine example_withState__ctor_fi(self, exception)
        implicit none
        integer*8 self, exception
C       DO-NOT-DELETE splicer.begin(example.withState._ctor)
        integer*8 statearray, logarray, dblarray
        call sidl_opaque__array_create1d_f(2, statearray)
        call sidl_bool__array_create1d_f(3, logarray)
        call sidl_double__array_create1d_f(2, dblarray)
        if ((statearray .ne. 0) .and. (logarray .ne. 0) .and.
     $       (dblarray .ne. 0)) then
           call sidl_opaque__array_set1_f(statearray, 0, logarray)
           call sidl_opaque__array_set1_f(statearray, 1, dblarray)
        else
C          a real implementation would not leak memory like this one
           statearray = 0
        endif
        call example_withState__set_data_f(self, statearray)
C       DO-NOT-DELETE splicer.end(example.withState._ctor)
        end
```

Of course, it is up to your application make the association between elements of the arrays and particular state variables. For example, you could say that element 0 of the double array is the kinematic viscosity and element 1 could be the airspeed velocity of an unladen swallow. Element 0 of the boolean array could specify African (true) or European (false). The destructor for this class could look something like this:

Fortran 77
```
        subroutine example_withState__dtor_fi(self, exception)
        implicit none
        integer*8 self, exception
```

```
C       DO-NOT-DELETE splicer.begin(example.withState._dtor)
        integer*8 statearray, logarray, dblarray
        call example_withState__get_data_f(self, statearray)
        if (statearray .ne. 0) then
           call sidl_opaque__array_get1_f(statearray, 0, logarray)
           call sidl_opaque__array_get1_f(statearray, 1, dblarray)
           call sidl_bool__array_deleteRef_f(logarray)
           call sidl_double__array_deleteRef_f(dblarray)
           call sidl_opaque__array_deleteRef_f(statearray)
C       the following two lines are not strictly necessary
           statearray = 0
           call example_withState__set_data_f(self, statearray)
        endif
C       DO-NOT-DELETE splicer.end(example.withState._dtor)
        end
```

In this example, an accessor function for the airspeed velocity of an unladen swallow could be implemented as follows:

Fortran 77

```
        subroutine example_withState_getAirspeedVelocity_fi(
     $     self, velocity, exception)
        implicit none
        integer*8 self, exception
        real*8 velocity
C       DO-NOT-DELETE splicer.begin(example.withState.getAirspeedVelocity)
        integer*8 statearray, dblarray
        call example_withState__get_data_f(self, statearray)
        if (statearray .ne. 0) then
           call sidl_opaque__array_get1_f(statearray, 1, dblarray)
           call sidl_double__array_get1_f(dblarray, 1, velocity)
        endif
C       DO-NOT-DELETE splicer.end(example.withState.getAirspeedVelocity)
        end
```

Chapter 9

FORTRAN 90 Bindings

Contents

9.1 Introduction

This chapter provides an introduction to the FORTRAN 90 bindings supported by Babel. Software written in FORTRAN 90 that illustrates both the caller, or client, side as well as the callee, or server side, is provided.

For ease of comparison, this chapter is patterned after the chapter on FORTRAN 77 bindings. Further, the initial support described below is very similar to that provided for FORTRAN 77.

9.2 Basic Types

The mapping for simple SIDL types to FORTRAN 90 is given in Table 9.1. For opaque pointers, the equivalent of a SIDL double is used. That is, the intermediate object reference assumes a 64-bit integer is used to enable portability between systems with a 32 bit address space and those with a 64 bit address space. On a 32 bit system, the upper 32 bits of these quantities are ignored. Systems with more than 64-bit pointers aren't currently supported.

The kind parameters used in Table 9.1 are defined in the F90 module `sidl`. The `sidl` modules defines integer parameters for `sidl_int`, `sidl_long`, `sidl_float`, `sidl_double`, `sidl_fcomplex`, `sidl_dcmoplex`, `sidl_enum` and `sidl_opaque` to give the appropriate type sizes to match the corresponding SIDL types.

For interfaces, classes and arrays, there is a derived type that holds an opaque pointer. The derived type for arrays of numeric types also has a F90 pointer to an array to provide native array access without function calls. For each interface and class, there are two modules created. In the first module, the derived type for the object and array are defined. In the second, the methods for the object/interface and arrays of the object/interface are defined. Clients of a class or interface, typically `use` the module containing the methods, and it in turn uses the module containing the types.

Generally, clients should treat opaque, interface, class and array values as black boxes. However, there is one value that is special. A value of zero for any of these quantities indicates that the value does not refer to an object. Zero is the equivalent of NULL. Any nonzero value is or should be a valid object reference. The method module provides functions to test whether an interface, class or array value `is_null` or is `not_null`. There is also a subroutine

Table 9.1: SIDL to FORTRAN 90 type mapping

SIDL TYPE	FORTRAN 90 TYPE
int	INTEGER (kind=sidl_int)
long	INTEGER (kind=sidl_long)
float	REAL (kind=sidl_float)
double	REAL (kind=sidl_double)
bool	LOGICAL
char	CHARACTER (LEN=1)
string	CHARACTER (LEN=*)
fcomplex	COMPLEX (kind=sidl_fcomplex)
dcomplex	COMPLEX (kind=sidl_dcomplex)
enum	INTEGER (kind=sidl_enum)
opaque	INTEGER (kind=sidl_opaque)

to initialize the value to indexFORTRAN 90!set_nullset_null. Clients should generally initialize new interface or class values to NULL.

The SIDL string type mapping is currently identical to that of the FORTRAN 77 mapping. That is, all FORTRAN 90 strings have a limited fixed size. When implementing a subroutine with an out parameter, the size of the string is limited to 512 characters. This can be changed when configuring babel by changing the value of SIDL_F90_STR_MINSIZE in runtime/sidl/babel_config.h before compiling and installing babel.

Enumerated types are just integer values. The integer parameter values are defined in a module.

9.3 Calling Methods From FORTRAN 90

All SIDL methods are implemented as FORTRAN 90 subroutines regardless of whether they have a return value or not. For object methods, the object or interface pointer is passed as the first argument to the subroutine before all the formally declared arguments. The exception is static methods, where the object or interface pointer does not appear in the argument list at all.

When a method has a return value, a variable to hold the return value should be passed as an argument following the formally declared arguments.

All methods can potentially throw exceptions, and the exception pointer is passed as an extra argument after the return value argument (if any). The exception argument behaves like an *out* parameter. A non-zero value indicates that an exception has been thrown. Exception handling for FORTRAN 90 is covered in greater depth in Section 9.4.

The name of the module that holds the method definitions is derived from the fully qualified name of the class or interface. You can generate the module name by replacing all the periods with underscores. For example, the methods for *sidl.SIDLException* are defined in a module named sidl_SIDLException in the file sidl_SIDLException.F90. The name of the module holding the derived type of the class or interface is the same as the one holding the methods except that it has _type appended to it. The types for *sidl.SIDLException* are called sidl_SIDLException_t and sidl_SIDLException_a, for the array, and they are defined in the file sidl_SIDLException_type.F90.

The name of the subroutine that FORTRAN 90 clients is the method's full name from the SIDL description. If the method is specified as overloaded (i.e., has a name extension), the method's full name will be used. That is, the concatenation of the short name and the name extension will be used for a unique method name.

For example, to call the deleteRef() method on a *SIDL.BaseInterface* interface, you would write:

Fortran 90

```
use sidl_BaseInterface
type(sidl_BaseInterface_t) :: interface1, interface2
type(sidl_BaseInterface_t) :: exception
logical                    :: areSame
!
! code to initialize interface1 & interface 2 here
!
```

```
  call deleteRef(interface1, exception)
```

In these examples, we are not checking the `exception` *out* parameter to see if an exception was thrown. Exception handing is covered below in Section 9.4. To call the isSame method on a *sidl.BaseInterface*, you would write:

Fortran 90

```
  use sidl_BaseInterface
  ! later in the code
  call isSame(interface1, interface2, areSame, exception)
  ! areSame holds the return value
```

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 5.6. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

Fortran 90

```
  use sidl
  use Overload_Sample
  type(Overload_Sample_t)          :: t
  type(sidl_BaseInterface_t)       :: exception
  logical                          :: b1, bretval
  integer (kind=sidl_int)  :: i1, iretval

  call new(t, exception)

  call getValue (t, iretval, exception)
  call getValueInt (t, i1, iretval, exception)
  call getValueBool (t, b1, bretval, exception)
```

Here is an example of what Babel will produce for an enumerated type with some of the whitespace and comments reduced for brevity.

Fortran 90

```
! File:          enums_car.F90
! Symbol:        enums.car-v1.0
! Symbol Type:   enumeration
! Babel Version: 0.8.2
! Description:   Client-side module for enums.car

module enums_car
! Symbol "enums.car" (version 1.0)
  use sidl

  integer (kind=sidl_enum), parameter :: porsche = 911
  integer (kind=sidl_enum), parameter :: ford = 150
  integer (kind=sidl_enum), parameter :: mercedes = 550
end module enums_car
```

For interfaces and classes, there is an implicit method called `cast()`. There are actually a set of overloaded methods that support every allowable cast between a type an all its parent types (objects and interfaces). The first argument is the object/interface to be cast, and the second argument is a variable of the desired type. If the value of the second argument after the call is `not_null`, the cast was successful. If the cast is successful, the caller owns the returned reference. In Babel releases prior to 0.11.0, `cast()` did not increment the reference count; now it does.

For non-abstract classes, there is an implicit method called `new()`. It creates and returns an instance of the class.

Here are examples of the use of these two methods:

Fortran 90

```
  use sidl_BaseClass
  use sidl_BaseInterface
```

```
type(sidl_BaseClass_t)     :: object
type(sidl_BaseInterface_t) :: interface
type(sidl_BaseInterface_t) :: exception
! perhaps other code here
call new(object, exception)
call cast(object, interface, exception)
```

Here is an example call to the `addSearchPath()`, a static method, in the *sidl.Loader* class:

Fortran 90

```
use sidl_Loader
use sidl_BaseInterface
type(sidl_BaseInterface_t) :: exception
! later
call addSearchPath('/try/looking/here', exception)
```

Your FORTRAN 90 must manage any object references created by the calls you make.

9.4 Catching and Throwing Exceptions in Fortran 90

When a method can throw an exception (i.e., its SIDL definition has a throws clause), a variable should be passed to hold an exception. For maximum backward compatibility, the exception argument type is a *sidl.BaseInterface* pointer that is assumed to implement the *sidl.BaseException* interface. The exception argument should appear after the return value when both occur in a method, and it behaves like an `out` parameter. After the call, the client should test this argument using `is_null` or `not_null`. If it is `not_null`, an exception was thrown by the method, and the method should respond appropriately. When an exception is thrown, the value of all other arguments is undefined, and the best course of action is to ignore their values. If your code does not check the exception argument after each call that can throw an exception, any exceptions that are thrown will be utterly ignored; they will not propagate automatically to higher level routines.

Here is another example adapted from the Babel regression tests. Package ExceptionTest has a class named Fib with a method declared in SIDL as follows:

SIDL

```
int getFib(in int n, in int max_depth, in int max_value, in int depth)
    throws NegativeValueException, FibException;
```

Here is the outline of a FORTRAN 90 code fragment to use this method.

Fortran 90

```
use sidl
use ExceptionTest_Fib
use ExceptionTest_FibException
use ExceptionTest_NegativeValueException
use sidl_BaseInterface
type(ExceptionTest_Fib_t)                    :: fib
type(sidl_BaseInterface_t)                   :: except, except2
type(ExceptionTest_FibException_t)           :: fibexcept
type(ExceptionTest_NegativeValueException_t) :: nvexcept
integer (kind=sidl_int)  :: index, maxdepth, maxval, depth, result
call new(fib, except)

index    = 4
maxdepth = 100
maxvalue = 32000
depth    = 0
call getFib(fib, index, maxdepth, maxvalue, depth, result, except)
if (not_null(except)) then
  call cast(except, fibexcept, except2)
```

```
    if (not_null(fibexcept)) then
!      do something here with the FibException
       call deleteRef(fibexcept, except2)
    else
      call cast(except, nvexcept, except2)
!     do something here with the NegativeValueException
      call deleteRef(nvexcept, except2)
    endif
    call deleteRef(except, except2)
  else
    write (*,*) 'getFib for ', index, ' returned ', result
  endif
  call deleteRef(fib, except2)
```

Here is an example of an implementation subroutine that throws an exception. Note you must `cast` the returned exception object into the exception `out` parameter. The `setNote` method provides a useful error message, and the `add` method helps provide a multi-language traceback capability (provided each layer of the call stack calls `add`).

Fortran 90

```
recursive subroutine ExceptionTest_Fib_getFib_mi(self, n, max_depth, &
  max_value, depth, retval, exception)
  use sidl
  use sidl_BaseInterface
  use ExceptionTest_Fib
  use ExceptionTest_NegativeValueException
  use ExceptionTest_FibException
  use ExceptionTest_Fib_impl
  ! DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib.use)
  use ExceptionTest_TooBigException
  use ExceptionTest_TooDeepException
  ! DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib.use)
  implicit none
  type(ExceptionTest_Fib_t) :: self
  integer (kind=sidl_int) :: n, max_depth, max_value
  integer (kind=sidl_int) :: retval, depth
  type(sidl_BaseInterface_t) :: exception
  type(sidl_BaseInterface_t) :: except2
! DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib)
  type(ExceptionTest_NegativeValueException_t) :: negexc
! ...lines deleted...
  character (len=*) myfilename
  parameter(myfilename='ExceptionTest_Fib_Impl.f')
  retval = 0
  if (n .lt. 0) then
     call new(negexc, except2)
     if (not_null(negexc)) then
        call setNote(negexc, &
             'called with negative n', except2)
        call add(negexc, myfilename, 57, &
                 'ExceptionTest_Fib_getFib_impl', except2)
        call cast(negexc, exception, except2)
        call deleteRef(negexc, except2)
        return
     endif
  else
! ...numerous lines deleted....
```

```
! DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib)
end subroutine ExceptionTest_Fib_getFib_mi
```

Please note that when your code throws an exception it should `deleteRef` any references it was planning to return to its caller. Any caller of a method that returns an exception should ignore the values of `out` and `inout` parameters, so anything you do not free will become a reference and memory leak. In general, it is good practice to call `set_null` on all `out` and `inout` array, class and interface arguments before returning when throwing an exception. This makes things work out better for clients who forget to check if an exception occurred or willfully choose to ignore it.

9.5 Invoking Babel to Generate F90 Stubs

Here is how you should invoke Babel to create the FORTRAN 90 stubs for an IDL file [1].

```
% babel --exclude-external --client=f90 file.sidl
```

or simply

```
% babel -E -c=f90 file.sidl
```

This will create a babel.make file, numerous C headers, numerous C source files, and some FORTRAN 90 files. The files ending in `_fStub.c` are called by the FORTRAN 90 module which in turn allow FORTRAN 90 to call a SIDL method. The files ending in `_type.F90` contain derived type definitions for classes and interfaces., and the other files ending in `.F90` are FORTRAN 90 modules containing methods.

You will need to compile and link the files ending in `_fStub.c` (i.e., `STUBSRCS` in `babel.make`) and all the files ending in `.F90` (i.e., `STUBMODULESRCS` and `TYPEMODULESRCS` in `babel.make`) into your application. Normally, the IOR files (`_IOR.c`) are linked together with the implementation file, so you probably don't need to compile them.

9.6 Implementing Classes in FORTRAN 90

Much of the information from the previous section is pertinent to implementing a SIDL class in FORTRAN 90. The types of the arguments are as indicated in Table 9.1. Your implementation can call other SIDL methods in which case follow the rules for client calls.

You should invoke Babel:

```
% babel --exclude-external --server=f90 file.sidl
```

or simply

```
% babel -E -s=f90 file.sidl
```

This will create a babel.make, numerous C headers, numerous C source files and some FORTRAN 90 source files. Your job is to fill in the FORTRAN 90 source files with the implementation of the methods. The files you need to edit all end with `_Impl.F90` and `_Mod.F90`. All your changes to the file should be made between code splicer pairs. Code between splicer pairs is retained by subsequent invocations of Babel; code outside splicer pairs is not.

Here is an example of the standard code splicer pairs in generated FORTRAN 90 code. You would replace the comment "Insert-Code-Here" associated with the "miscellaneous code start" splicer pair with code needed for your implementation such as additional abbreviation file(s) and any local, or private, subroutines. For the subroutine's "use" splicer pair, you would replace the "Insert-Code-Here {Pkg.Class.name.use} (use statements)" comment with any use statements that are needed by the subroutine. Finally, you would add the implementation between the subroutine body's splicer pairs in the place of the "Insert-Code-Here {Pkg.Class.name} (name method)" comment.

[1] For information on additional command line options, refer to Section 3.2.

Fortran 90

```
! DO-NOT-DELETE splicer.begin(_miscellaneous_code_start)
! Insert-Code-Here {_miscellaneous_code_start} (extra code)
! DO-NOT-DELETE splicer.end(_miscellaneous_code_start)

.
.
.

recursive subroutine Pkg_Class_name_mi(self, arg, exception)
  use sidl
  use sidl_BaseInterface
  use sidl_RuntimeException
  use Pkg_Class
  use Pkg_Class_impl
  ! DO-NOT-DELETE splicer.begin(Pkg.Class.name.use)
  ! Insert-Code-Here {Pkg.Class.name.use} (use statements)
  ! DO-NOT-DELETE splicer.end(Pkg.Class.name.use)
  implicit none
  type(Pkg_Class_t) :: self ! in
  integer (kind=sidl_int) :: arg ! in
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(Pkg.Class.name)
! Insert-Code-Here {Pkg.Class.name} (name method)
! DO-NOT-DELETE splicer.end(Pkg.Class.name)
end subroutine Pkg_Class_name_mi
```

Each `_Impl.F90` file contains numerous partially implemented subroutines. The `SUBROUTINE` and `END SUBROUTINE` statements have been generated and the types of the arguments declared. As mentioned above, you must provide any needed use statements and the body of each subroutine to implement the expected behavior of the method.

There are two implicit methods (i.e., methods that did not appear in the SIDL file) that must also be implemented if the object is to have state (i.e., data associated with the instance). The `_ctor()` method is a constructor function that is run whenever an object is created. The `_dtor()` method is a destructor function that is run whenever an object is destroyed. If there is not state then these functions are typically empty.

The SIDL IOR keeps a pointer for each object that is intended to hold a pointer to the object's internal data. The FORTRAN 90 skeleton provides two functions that the FORTRAN 90 developer will need to use to access the private pointer. The name of the function is derived from the fully qualified type name by replacing periods with underscores and appending `__get_data_m` or `__set_data_m`. The first argument is the object pointer (i.e., self), and the second is a derived type defined in the `_Mod.F90` file. Here is an excerpt from a `_Mod.F90` file for an object whose state requires a single integer value.

Fortran 90

```
#include "sort_SimpleCounter_fAbbrev.h"
module sort_SimpleCounter_impl

! DO-NOT-DELETE splicer.begin(sort.SimpleCounter.use)
use sidl
! DO-NOT-DELETE splicer.end(sort.SimpleCounter.use)

type sort_SimpleCounter_priv
  sequence
  ! DO-NOT-DELETE splicer.begin(sort.SimpleCounter.private_data)
   integer(kind=sidl_int) :: count
  ! DO-NOT-DELETE splicer.end(sort.SimpleCounter.private_data)
end type sort_SimpleCounter_priv
```

```
type sort_SimpleCounter_wrap
  sequence
  type(sort_SimpleCounter_priv), pointer :: d_private_data
end type sort_SimpleCounter_wrap

end module sort_SimpleCounter_impl
```

The derived type `sort_SimpleCounter_private` is the type where the developer adds data to store the object's state, and `sort_SimpleCounter_wrap` exists simply to facilitate transferring the pointer to a `sort_SimpleCounter_private` to and from the IOR.

Typically for a class with state, the developer needs to `allocate(pd%d_private_data)` in the constructor, `_ctor`, and `deallocate(pd%d_private_data)` in the destructor, `_dtor`. Here is a concrete example of a constructor.

Fortran 90

```
recursive subroutine sort_SimpleCounter__ctor_mi(self, exception)
  use sidl
  use sidl_BaseInterface
  use sidl_RuntimeException
  use sort_SimpleCounter
  use sort_SimpleCounter_impl
  ! DO-NOT-DELETE splicer.begin(sort.SimpleCounter._ctor.use)
  ! Insert-Code-Here {sort.SimpleCounter._ctor.use} (use statements)
  ! DO-NOT-DELETE splicer.end(sort.SimpleCounter._ctor.use)
  implicit none
  type(sort_SimpleCounter_t) :: self ! in
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(sort.SimpleCounter._ctor)
  type(sort_SimpleCounter_wrap) :: dp
  allocate(dp%d_private_data)
  dp%d_private_data%count = 0
  call sort_SimpleCounter__set_data_m(self, dp)
! DO-NOT-DELETE splicer.end(sort.SimpleCounter._ctor)
end subroutine sort_SimpleCounter__ctor_mi
```

Here is the corresponding destructor.

Fortran 90

```
recursive subroutine sort_SimpleCounter__dtor_mi(self, exception)
  use sidl
  use sidl_BaseInterface
  use sidl_RuntimeException
  use sort_SimpleCounter
  use sort_SimpleCounter_impl
  ! DO-NOT-DELETE splicer.begin(sort.SimpleCounter._dtor.use)
  ! Insert-Code-Here {sort.SimpleCounter._dtor.use} (use statements)
  ! DO-NOT-DELETE splicer.end(sort.SimpleCounter._dtor.use)
  implicit none
  type(sort_SimpleCounter_t) :: self ! in
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(sort.SimpleCounter._dtor)
  type(sort_SimpleCounter_wrap) :: dp
  call sort_SimpleCounter__get_data_m(self, dp)
  deallocate(dp%d_private_data)
! DO-NOT-DELETE splicer.end(sort.SimpleCounter._dtor)
end subroutine sort_SimpleCounter__dtor_mi
```

9.7 Accessing SIDL Arrays From FORTRAN 90

SIDL r-arrays are passed to and from methods as normal FORTRAN 90 arrays. You do not need to include the index variables because the values are determined from the FORTRAN 90 array extents in each dimension.

The client-side interface for the `solve` example introduced in Section 5.4 behaves as if it is a FORTRAN 77 function with the following overloaded interface:

Fortran 90

```
private :: solve_1s, solve_2s
interface solve
  module procedure solve_1s, solve_2s
end interface

recursive subroutine solve_1s(self, A, x, exception)
  implicit none
  ! in num.Linsol self
  type(num_Linsol_t) , intent(in) :: self
  ! in array<double,2,column-major> A
  type(sidl_double_2d) , intent(in) :: A
  ! inout array<double,column-major> x
  type(sidl_double_1d) , intent(inout) :: x
  ! out sidl.BaseInterface exception
  type(sidl_BaseInterface_t) , intent(out) :: exception
end subroutine solve_1s

recursive subroutine solve_2s(self, A, x, exception)
  implicit none
  ! in num.Linsol self
  type(num_Linsol_t) , intent(in) :: self
  ! in rarray<double,2> A(m,n)
  real (kind=sidl_double) , intent(in), dimension(:, :) :: A
  ! inout rarray<double> x(n)
  real (kind=sidl_double) , intent(inout), dimension(:) :: x
  ! out sidl.BaseInterface exception
  type(sidl_BaseInterface_t) , intent(out) :: exception
  ! in int m
  integer (kind=sidl_int) :: m
  ! in int n
  integer (kind=sidl_int) :: n
end subroutine solve_2s
```

You can use either normal FORTRAN 90 arrays or normal SIDL arrays when calling a FORTRAN 90 method, but you cannot use a mixture.

The server-side interface for `solve` is similar. Note, the lower index each dimension of every incoming array is always zero.

Fortran 90

```
recursive subroutine num_Linsol_solve_mi(self, A, x, m, n, exception)
  use sidl
  use sidl_BaseInterface
  use sidl_RuntimeException
  use num_Linsol
  use sidl_double_array
  use num_Linsol_impl
  ! DO-NOT-DELETE splicer.begin(num.Linsol.solve.use)
  ! Insert-Code-Here {num.Linsol.solve.use} (use statements)
  ! DO-NOT-DELETE splicer.end(num.Linsol.solve.use)
```

```
  implicit none
  type(num_Linsol_t) :: self ! in
  integer (kind=sidl_int) :: m ! in
  integer (kind=sidl_int) :: n ! in
  type(sidl_BaseInterface_t) :: exception ! out
  real (kind=sidl_double), dimension(0:m-1, 0:n-1) :: A ! in
  real (kind=sidl_double), dimension(0:n-1) :: x ! inout

! DO-NOT-DELETE splicer.begin(num.Linsol.solve)
! Insert-Code-Here {num.Linsol.solve} (solve method)
! DO-NOT-DELETE splicer.end(num.Linsol.solve)
end subroutine num_Linsol_solve_mi
```

For normal SIDL arrays, the normal SIDL C function API is available from FORTRAN 90 to create, destroy, and access array elements and meta-data. The array routines are in a module. For *sidl.SIDLException*, the array module is named sidl_SIDLException_array, and the array module is defined in the sidl_SIDLException_array.F90.

For SIDL types dcomplex, double, fcomplex , float, int, and long, SIDL provides an array pointer to get direct access to the array elements. For the other types, you must use the functional API to access array elements.

The SIDL derived type for a SIDL array is named after the class, interface or basic type that it holds and the dimension of the array. For *sidl.SIDLException*, the array derived types are named sidl_SIDLException_1d, sidl_SIDLException_2d, sidl_SIDLException_3d, ... up to sidl_SIDLException_7d. For the basic types, they are treated as *sidl.dcomplex*, *sidl.double*, *sidl.fcomplex*, etc. Each of these derived types has a 64 bit integer to hold an opaque pointer.

The derived type for SIDL types dcomplex, double, fcomplex , float, int, and long also has a pointer to an array of the appropriate type and dimension. For example, here is the derived type for 2d and 3d arrays of doubles.

Fortran 90

```
  use sidl
  type sidl_double_2d
    sequence
    integer (kind=sidl_arrayptr) :: d_array
    real (kind=sidl_double), pointer, &
      dimension(:,:) :: d_data
  end type sidl_double_2d

  type sidl_double_3d
    sequence
    integer (kind=sidl_arrayptr) :: d_array
    real (kind=sidl_double), pointer, &
      dimension(:,:,:) :: d_data
  end type sidl_double_3d
```

You can access the array with the F90 array pointer d_data just like any other F90 array. However, you *must not* use the F90 builtins allocate or deallocate on d_data. You must use SIDL functions, createCol, createRow, create1d, create2dRow, or create2dCol, to create a new array. These SIDL routines initialize d_data to refer to the data allocated in d_array. Note, create1d, create2dRow, and create2dCol create arrays whose lower index is *0* not 1. To create arrays with a lower index of 1, use createCol or createRow.

You can call things like LINPACK or BLAS if you want, but you should check the stride to make sure the array is packed as needed. You can check stride(i), which indicates the distance between elements in dimension i. A value of 1 means elements are packed densely in dimension i. Negative stride values are possible. When an array is sliced, the resulting array might not even have one densely packed dimension.

In F90, generic arrays are represented as the derived type sidl__array. Note there are two underscores in sidl__array. The derived type is defined in the sidl_array_type module, and the generic array subroutines are defined in the sidl_array_array module which declared: addRef, deleteRef, dimen, type, isColumnOrder, isRowOrder, is_null, no_null, set_null, lower, upper, length, stride, and smartCopy. The F90 binding includes a cast subroutine to can from a generic array to an array of a particular type

and dimension and vice versa.

Chapter 10

Java Bindings

Contents

10.1 Introduction

This chapter provides an introduction to the Java bindings for SIDL, including illustrations of both callers and callees written in Java. It shows how to use Babel to wrap the implementation of software written in Java as well as how to call software, possibly implemented in any other supported language, from Java.

10.2 Basic Types

Most SIDL types map directly into Java as shown in Table 10.1.

10.3 Client Side: Using SIDL Classes and Methods

SIDL's object model is very similar to Java's, and therefore maps easily into Java's object model. A SIDL object is treated almost exactly the same in Java as any other Java object, the only difference being that all data held by the object is private, and all methods are public.

Importing SIDL packages and classes is also exactly the same as in Java. For example, assume there is a package `test` that includes the class `HelloWorld`, and you wish to print this message in your program. The following code segment does this.

Table 10.1: SIDL to Java Type Mappings

SIDL TYPE	JAVA TYPE
int	int
long	long
float	float
double	double
bool	boolean
char	char
string	String
fcomplex	FloatComplex
dcomplex	DoubleComplex
enum	Enum
opaque	long
interface	interface
class	class
array	type.Array

Java

```
import test.HelloWorld;

public static main(String args[]) {

  HelloWorld hi = new HelloWorld();
  hi.printMsg();
}
```

Writing the fully qualified class name would also have sufficed. `test.HelloWorld hi = new test.HelloWorld()`

Babel also generates Java code in line with Java's use of directories to organize packages and classes as files. For example, assume you are generating babel code in a directory named `babelcode`. Assume your package `test` contains 2 classes `HelloWorld` and `GoodbyeWorld`. After running `babel -cJava test.sidl` you will have a new directory in `babelcode` named `test` which will contain 2 files, `HelloWorld.java` and `GoodbyeWorld.java`. These classes will be accessible from your Java program as long as `babelcode` is in your `CLASSPATH`.

10.4 Server Side: Writing SIDL classes in Java

Babel also supports calls to SIDL classes implemented in Java. These classes obey the same rules as the client side Java classes, except that is this case the file, class, and method names all end in `_Impl`.

As is the case with other Babel server side files, only the code written between splicer blocks will be preserved between calls of Babel. Make sure any data and code is kept in the designated areas, otherwise it won't be there after you run Babel on those files.

Another interesting fact of the Server Side is that it inherits from the Client Side Java class. This allows us to call local methods directly. Take this recursive Fibonacci function implementation for example:

Java

```
class Fib_Impl extends Fib {
  public int getFib_Impl(int x) {
    // DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib)
    if(x >= 2) {
      return getFib(x-1) + getFib(x-2);
    } else {
      return 1;
    }
    // DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib)
```

```
  }
}
```

Here the client side class is name `Fib`, and therefore the Server Side class is `Fib_Impl`. The same relation is true for the `getFib` method. You can also see that we are able to call `getFib`, the client side method, directly. A call like this will go through Babel glue code, as it should. You should not try to make calls directly to `_Impl` methods. It won't work at all on different objects, and it breaks the object model if used on methods in the current object. (That is, it is possible to call `foo_Impl` in the current object, but because the call will not go through Babel, any inheritance information will be lost, and the wrong version of the method may be called. Simply call `foo` in the standard way.)

This also means there is no way to have Server Side object inherit from non SIDL Java classes, in fact, there are no splicer blocks available for inheritance, so implementing interfaces on the Server Side is also not supported. This is because we feel that having the Server side inherit from non-SIDL classes is probably not a good idea.

10.5 Casting Objects

In some cases it is necessary to cast the internal representation of an object as well as the Java object. (For example, getting an object from a SIDL array of superclass objects.) In these cases a Java cast is insufficient. Therefore we have provided two casting functions.

`_cast(object)` is a static function included with every SIDL class that returns object passed in to cast that class. For example, in order to cast an object of type `sidl.BaseClass` to `foo.Bar` simply write `foo.Bar newobj = (foo.Bar) foo.Bar._cast(oldobj)`. If this is an invalid cast, `_cast` will return `null`.

The alternative is `_cast2(``ClassName'')`. This is a cast function that is included with every SIDL object. It performs basically the same function as `_cast`, but the form is `object._cast2(``ClassName'')`. It takes a fully qualified class name. If the cast is invalid, or a class of that name cannot be found, this function returns `null`.

Both of these functions return a `sidl.BaseClass` which then must be Java casted to the correct Java class type. Also, in casting, they both create a new Java object that owns a new reference to the IOR object. In Java you never have to worry about reference counting, but this does mean that the pre-cast object still exists and is valid.

10.6 Out and Inout arguments

In C or C++ out and inout arguments are handled by passing pointers to the data so that if the data is changed, the pointer will be pointing to the new, correct, data. Because Java does not support pointers, each SIDL type and class has a static inner `Holder` class. This Holder class can hold a single variable or object of the correct type. There are functions `get()` and `set()` for getting or setting this object.

10.7 Using SIDL arrays with Java

Every object and type defined in SIDL can be put into a SIDL array of that type. Arrays are a fairly complex topic, and the specifics of the Babel Array API are discussed earlier in Section 5.4. Suffice to say that the entire API is available in Java, except for

`ensure`, `borrow`, and `first`, all of which have no real use in Java. `addRef` and `deleteRef` exist in Java, but shouldn't be used, because the Java decrements the reference count itself when it garbage collects a SIDL object or array. If it is necessary to `deleteRef` an array, you should use the `destroy()` array function instead.

More to the point are the specifics of the Java implementation. Each SIDL type and class includes a static inner class named `Array`. This is the main Array class, and in order to support up to 7 dimensional arrays, every method takes either 7 array indices, or an array of indices. For example, in order to get the element (2,3) from a 2 dimensional array, we would type `arry._get(2,3,0,0,0,0,0)`.

Since typing all those zeros can get a little tedious, we also implemented a set of subclasses of Array. One subclass for each dimension. So, if we had and `Array2` instead of an `Array` we could simply type `arry2._get(2,3)` to get the correct element.

These numbered Array subclasses improve on the Array API usability somewhat, but that do have a side effect. In order to avoid conflicts between the Array superclass and the numbered Array subclass functions, all other basic

Array methods found in the Array superclass are preceded by an underscore '_'. For example, in order to get an array's dimension, you can type `arry._dim()`. The numbered arrays all inherit these methods, so `arry2._dim()` will also work, although in this case, the answer should be obvious.

Furthermore, there is another underscore rule for Arrays in Java. All numbered arrays have two `get` and two `set` functions. The `_get` and `_set` functions are the same in Array and all the Array# subclasses, they simply pass the arguments of the `_get` call down to the underlying implementation. However, the underscore-less `get` and `set` do bounds checking in Java before calling the underlying implementation, and, if there is a problem, throw an `ArrayIndexOutOfBoundsException`.

Because the numbered arrays are subclasses of `Array`, if necessary you can Java cast an `Array#` to an `Array`. However, some functions return an `Array`. In order to convert an `Array` to the correctly numbered array, we provided a function in `Array` called `_dcast()`. In order to cast an `Array` object to a numbered array, simply call `_dcast()` on it. For example, assume we have a 1 dimensional array of type `foo.Bar` called `arry` that is represented by the Java class `Array`. In order to get a correctly numbered array type:

```
foo.Bar.Array1 arry1 = arry._dcast();
```
Java

After this cast we have 2 references to the same array, `arry` and `arry1`.

Finally, the Java array constructors are slightly different then they are in other languages. This is the constructor definition for `Array`.

```
public Array(int dim, int[] lower, int[] upper, boolean isRow)
```
Java

This constructor creates and array of dimension `dim`. It takes two arrays of integers to define the lower and upper bounds of each dimension in the array. If lower or upper has fewer elements than there are dimensions in the array, or any element in lower is larger than the corresponding element in upper, this constructor will throw an exception. Finally, this constructor takes a boolean `isRow`. If `isRow` is true, this constructor will create a SIDL array in row-major order, if it is false, it will create an array in column-major order.

The constructors for numbered arrays are simpler. Here's the constructor for a 2 dimensional array:

```
public Array2( int l0, int l1, int u0, int u1, boolean isRow)
```
Java

The dimension argument is no longer necessary, and it is no longer necessary to create arrays of bounds to pass into the constructor. `l0` and `l1` are the lower bounds. and `u0` and `u1` are the upper bounds. This constructor still includes the choice between column and row major orders.

If all your lower bounds are 0, you can use an even simpler constructor:

```
public Array2( int s0, int s1, boolean isRow)
```
Java

Another alternate way to construct sidl arrays is present in numbered arrays. The following constructor takes a Java 2 dimensional array, and copies it into a SIDL 2 dimensional array:

```
public Array2(foo.Bar[][] array, boolean isRow)
```
Java

If you already have a numbered SIDL array of the correct dimension, you can copy a java array into it with the method `fromArray`. The method takes the same arguments as the constructor above, and returns nothing.

If you wish to go the other way, to copy a sidl array into a Java array, you may use the numbered array function `toArray`. `toArray` takes no arguments, and returns a new Java array with the SIDL array elements copied into it.

10.8 Interfaces and Abstract Classes

Babel implements SIDL interfaces as Java interfaces in Java. This is a close mapping in general, but it does have the problem that Java interfaces can't hold data. Since we need the correct IOR pointer in order to place that interface in an array or throw it as an Exception, the lack of data becomes a problem. For this reason, we have created Wrapper classes for interfaces and abstract classes.

All interfaces and abstract classes have static inner class named `Wrapper`. This `Wrapper` class holds the interface IOR pointer, and also inherits from `gov.llnl.babel.BaseClass` and implements the outer interface. Therefore,

you can call all the interface methods on the wrapper object, as well as `gov.llnl.babel.BaseClass` methods such as `_cast2`, and `isType`.

This wrapper class is what is returned when an interface is gotten out of an array, a method takes or returns an interface, or when an exception implemented as an interface is caught. (There's actually a difference here. While what is gotten out of the Array or returned from a method is a Wrapper object, the programmer doesn't usually need to worry about that, as is shown in the example below. In the case of exceptions, you actually do have to catch the Wrapper. Exceptions are covered in more detail in Subsection 10.9) Because wrapper classes inherit only from an interface, they can be java casted to their enclosing interface, or it's super-interfaces, but must be Babel casted to any classes. In this example, Subclass implements Super-Interface:

Java
```
SuperInterface.Array1 arry = new SuperInterface.Array1(5, true);
SubClass obj = new SubClass();
arry.set(0, (SuperInterface)obj);
obj = null;
SuperInterface temp = arry.get(0);
obj = (SubClass) temp;    //INCORRECT Will throw ClassCastException

obj = (SubClass) SubClass._cast((SuperInterface.Wrapper)temp); //CORRECT
```

Sometimes you can get away with not Java casting the interface to the Wrapper class before Babel casting it, but not in general. (Usually you don't have to when the interface was gotten out of an array)

Here's an example of casting an interface on the server side:

Java
```
public objarg.SubClass toClass_Impl (/*in*/ objarg.Iface ifcy ) {
  // DO-NOT-DELETE splicer.begin(objarg.SubClass.toClass)
  objarg.SubClass ret =  (objarg.SubClass)
    ((objarg.Iface.Wrapper)ifcy)._cast2("objarg.SubClass");
  return ret;
  // DO-NOT-DELETE splicer.end(objarg.SubClass.toClass)
}
```

10.9 Exceptions

Exceptions are caught and thrown in exactly the same way as Java exceptions. If an exception is defined in SIDL, Babel will generate the code for it, and the exception can be thrown in Java. The only difference is that SIDL exception constructor cannot take a String. Instead, the message must be set with SIDL's `setNote` method, the message is gotten with SIDL's `getNote` method. This is important because SIDL exceptions inherit from the Java Class `Exception`. The Java compiler *will not* give an error if getMessage is called, but the message returned will not have been from SIDL.

The other problem is that regular Java exceptions cannot be passed on by Babel. Of course, it's not possible to throw normal non-SIDL exceptions from a SIDL Java function, the Java compiler will throw an error. (Unless you have changed the Java method "throws" statement outside the splicer blocks, which you should never do.) However, Java runtime exceptions, such as `ArrayIndexOutOfBoundsException` can be thrown. In this case, an error message and stack trace are printed to stderr, the method returns 0, the values of any out or inout arguments are set to NULL, and the program proceeds.

Finally, SIDL Exceptions may be interfaces, where as Java exceptions are always classes. This means Babel allows you to throw an interface. However, in Java we actually need to throw the interface's Wrapper class.

In this example we have a class `FibException` which implements two exception interfaces, `NegativeValueException` and `TooDeepException`. These two Exceptions are thrown by a babelized method named `getFib`. `getFib` is a standard recursive Fibonacci number generating function, in which if something goes wrong, it throws one of these two exceptions. First, server side:

```
public int getFib_Impl ( /*in*/ int n)
throws NegativeValueException.Wrapper, TooDeepException.Wrapper {
  if (n < 0) {
    FibException fex = new FibException();
    NegativeValueException.Wrapper neg = (NegativeValueException.Wrapper)
      NegativeValueException.Wrapper._cast(fex);
    neg.setNote("n negative");
    throw neg;
  }

  //  .... Do Fibonacci stuff ....
}
```
Java

You can see here some of the hoops you have to jump though to throw an interface. First, since we cannot create an interface, or it's Wrapper, directly, we first create a new `FibException` and cast it to the interface we want. Secondly, we have to refer to the Wrapper's full name in this case, because it is impossible to throw interfaces in Java. Finally, as with all SIDL Exceptions, we use `setNote` to set the exception's message, as we cannot pass in a message with the constructor.

Next the client side:

```
try {
  fib.getFib(-1);
} catch (NegativeValueException.Wrapper ex) {
  System.err.println(ex.getNote());
} catch (TooDeepException.Wrapper ex) {
  System.err.println(ex.getNote());
} catch (java.lang.Exception ex) {
  if (((sidl.BaseInterface)ex).isType("sidl.SIDLException")) {
    check(PASS, true, "Unexpected SIDL Exception thrown");
  } else {
    check(PASS, false, "Unexpected and unkown exception thrown");
  }
}
```
Java

In order to differentiate between the two different interfaces in this case we must catch the Wrappers explicitly by their fully qualified names. In the `exceptions` regression test we discover the types of the Exceptions by calling the SIDL function `isType` on them. However, because SIDL can cast between the two interfaces, in this case `isType` would return true no matter what the exception originally was. The final catch `java.lang.Exception ex` should not ever be executed in out example code. `getFib` does not throw any other kinds of exceptions, and babel cannot throw non-SIDL Exceptions. This was included because it demonstrates the most basic way to differentiate a SIDL exception from a Java exception.

10.10 Enumerations

Enumerations are implemented as `final static ints` in their own Java class, and as such, are accessed just like variables in that class. For example, if we had a sidl package named `dealership` that contained the following code segment:

```
  enum car {
    porsche = 911,
    ford = 150,
    mercedes = 550
  };
```
Java

we would be able to get the value assigned to a Porsche by typing `dealership.car.porsche`.

10.11 Invoking Babel to generate Java bindings

To create Java stubs (i.e. code to support Java clients to a set of SIDL classes or interfaces), you should invoke Babel as follows [1]:

```
% babel --exclude-external --client=Java file.sidl
```

or more cryptically

```
% babel -E -cJava file.sidl
```

This will create a great plethora of files, including a directory named `file`. This directory contains the Java client classes, if you want to take a look at them. The files ending in `_IOR.h` and `_IOR.c` are the Intermediate Object Representation. The files ending with `_jniStub.c` are the JNI stubs — the interface between a Java client and the IOR. The "jni" in the filename represents the fact that we use the Java Native Interface to communicate between Java and the IOR representation. The remaining header files have external Java API that Java clients may use.

To use the Java stubs, you must compile the stub files whose file names end with `_jniStub.c` and link them against the SIDL runtime library and a backend implementation. The resulting library needs to be referenced in a `.scl` file listed in the SIDL_DLL_PATH environment variable so that the Babel runtime library loader can find it. Also, the current directory needs to be in the CLASSPATH environment variable so that Java can find the `file` and `sidl` directories that contain the Java component of the client side.

10.12 Invoking Babel to generate Java implementations

To implement a set of SIDL classes in Java, you should invoke Babel as follows:

```
% babel --exclude-external --server=Java file.sidl
```

or use the short form

```
% babel -E -sJava file.sidl
```

The directory structure that results from this command is that same as the client side, there are just a bunch more files. In the `file` directory there are new files that end in `_Impl.java`. These are the java files where you should write your implementation. All of your methods in this class now also end in `_Impl`. In the current directory there are also new files that end in `_jniSkel.c`. These files are the equivalent to the `_jniStub.c` for the client side.

You should also notice that all the Client side files have been generated in addition to the new Server side files. These files are present to allow for calling methods on the current object in the Implementation java file. You can safely ignore them.

10.13 Environment Variables

There are some environment variables associated with running Java with Babel. You can find examples for some of these in the regression tests included with babel.

CLASSPATH: The CLASSPATH is an environment variable that Java uses to find `.class` files. It's is not specific to Babel, but it is necessary. It consists of a colon delimited series of directories to search for Java classes. In addition to any of your own Class files for use in Java server side, you should include `build dir/lib/sidl-ver.jar` where ver is the current sidl version, and `build dir/runtime/java`.

BABEL_JVM_FLAGS: This environment variable is used *only* when passing java command line variables to Java server side. It consists of a semi-colon delimited list of command line variables you wish to pass to Java server side. (A useful one might be -Xcheck:jni))Here's an example:

[1] For information on additional command line options, refer to Section 3.2.

```
BABEL_JVM_FLAGS="-verbose:gc;-Xmx500m"
```
shell

It is also necessary to set your LD_LIBRARY_PATH (or LIBPATH on AIX) and SIDL_DLL_PATH correctly. Not including all the necessary files in the SIDL_DLL_PATH and LD_LIBRARY_PATH *can* crash the JVM in unhelpful ways. Babel tries to generate helpful error messages, but sometimes the JVM crashes due to missing files and doesn't generate very helpful output. If the JVM crashes, make sure you've included all the necessary files in your SIDL_DLL_PATH and LD_LIBRARY_PATH.

Chapter 11

Python Bindings

Contents

Babel requires a Python shared library. Because Python 2.3 has a configure/build system that builds shared libraries on many architectures, we recommend that you use Python 2.3 or beyond.

11.1 How to Create a SIDL Object in Python

(once you've built the Python extension module)

You need to import the extension module and then calling a method to create an instance. If you have a class whose fully qualified name is x.y.z, you would say:

```
>>> import x.y.z
>>> obj = x.y.z.z()
```

The last part of the class name is repeated. You can also use **`from x.y.z import *`** if you prefer; although, you must guarantee that there are no namespace collisions.

In some cases, the Python extension module may be name `zmodule.so` instead of simply `z.so`. This might tempt you to say **`import x.y.zmodule`** instead of just **`import x.y.z`**; resist this temptation!

11.2 How to Cast SIDL Objects in Python

Let's say you have an object `obj`, and you would like to see if it is an instance of a SIDL class or interface whose fully qualified name is `x.y.z`. Here is how you do it.

```
>>> import x.y.z
>>> zobj = x.y.z.z(obj)
```

Of course, you don't need the import if you know that `x.y.z` has already been imported. If `zobj` is not equal to `None`, the cast was successful.

11.3 How to Call Methods from Python

Once you have created an object, you call methods on it using normal Python method calls. The arguments to the method only include the `in` and `inout` arguments, and the return value of the Python method includes the SIDL return value and the `inout` and `out` parameters. Hopefully, this will seem natural to Python programmers. For the following example, the object `obj` has a method `passeverywhere` with the following SIDL declaration:

SIDL
```
double passeverywhere( in double d1, out double d2, inout double d3 );
```

You can see the Python calling signature with **`print obj.passeverywhere.__doc__`**. Here is what that shows for this example:

Python
```
$ python
>>> import Args.Cdouble
>>> obj = Args.Cdouble.Cdouble()
>>> print obj.passeverywhere.__doc__
passeverywhere(in double d1,
               inout double d3)
RETURNS
   (double _return,
    out double d2,
    inout double d3)
```

In the method documentation, the SIDL method's return value is called `_return`; and unless the method is `void`, the return value always appears first. The fact that `_return` starts with an underbar should alert you to the fact that it is not a parameter because parameter names cannot start with an underbar. The document comments from the SIDL file (i.e. comments enclosed in `/** */` comments) appear below the Babel generated signature documentation.

Static methods of a class are available in the Python `x.y.z` namespace assuming you use the **`import x.y.z`** command. Static methods have documentation just like class methods.

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 5.6. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

Python
```
b1 = 1
i1 = 1

t = Overload.Sample.Sample()

nresult = t.getValue()
iresult = t.getValueInt(i1)
bresult = t.getValueBool(b1)
```

11.4 Catching and Throwing Exceptions in Python

Python exceptions must be Python classes; they cannot be a C extension type — the mechanism used to wrap SIDL objects as Python objects. Because of this, Babel defines an exception class for each SIDL type that implements `sidl.BaseException`. For a type called `x.y.z`, the Python exception class is named `x.y.z._Exception`. In Babel 0.10.2 and previous releases, the Python exception class was named `x.y.z.Exception`, but this name can

potentially collide with the class constructor or a static method named `Exception`. For backwards compatibility, Babel defines `x.y.z.Exception` if the name `Exception` is not used in the class.

SIDL exceptions are caught and thrown very much like normal Python exceptions are caught and thrown except you need to use the Python exception class for the SIDL type. The exception value holds the SIDL object as attribute `exception`. Here is an example of a code catching exceptions from a call to `getFib`. Note that `eobj.exception` is an instance of `ExceptionTest.NegativeValueException.NegativeValueException`, the Python type corresponding to the SIDL type `ExceptionTest.NegativeValueException`.

Python
```
try:
  fib.getFib(-1, 10, 10, 0)
except ExceptionTest.NegativeValueException._Exception:
  (etype, eobj, etb) = sys.exc_info()
  # eobj is the SIDL exception object
  print eobj.exception.getNote()  # show the exception comment
  print eobj.exception.getTrace() # and traceback
```

Here is an example of a Python implementation function that throws an exception. The `setNote` method provides a useful error message, and the `add` method helps provide a multi-language traceback capability (provided each layer of the call stack calls `add`).

Python
```
def getFib(self, n, max_depth, max_value, depth):
  # sidl EXPECTED INCOMING TYPES
  # ============================
  # int n, max_depth, max_value, depth
  #
  # sidl EXPECTED RETURN VALUE(s)
  # =============================
  # int _return
  # DO-NOT-DELETE splicer.begin(getFib)
  if (n < 0):
    ex = ExceptionTest.NegativeValueException.NegativeValueException()
    ex.setNote("n negative")
    ex.add(__name__, 0, "ExceptionTest.Fib.getFib")
    raise ExceptionTest.NegativeValueException._Exception, ex
  # numerous lines deleted
  # DO-NOT-DELETE splicer.end(getFib)
```

11.5 Building Python Extension Modules

SIDL creates a `setup.py` file that can be used to build the Python extension modules that you create. `setup.py` uses the Python distutils package to build the Python extension modules. There are two extra command line arguments.

- `--include-dirs=` — Use this to specify extra directories for the preprocessor include path. This is like `-I` for most C compilers.
- `--library-dirs=` — Use this to specific extra directories for static or shared libraries. This is like `-L` for most C compilers/loaders.

Normally, you need to specify the directory where the SIDL runtime headers and SIDL Python headers are stored with `--include-dirs=`. You also need to specify the directory where `libsidl.so` is stored. Here is a hypothetical example:

```
setup.py --include-dirs=/usr/local/include
--include-dirs=/usr/local/include/python
--library-dirs=/usr/local/lib build_ext --inplace
```

It is unlikely that any installation actually uses those settings.

11.6 Setting up to Run Python

Here I assume that you've installed Babel in directories rooted at `$PREFIX`. You need to have `$PREFIX/python` in your `PYTHONPATH` environment variable in addition to the directory where you are doing your work.

On many systems, you will need `$PREFIX/lib` in your `LD_LIBRARY_PATH` (or whatever system setting controls which directories are searched for shared libraries/dynamic link libraries).

You will likely need to use `SIDL_DLL_PATH` (a semicolon separated path) to provide the path to the directory that holds the shared library/dynamic link library containing the implementation of the SIDL objects.

11.7 Notes

The Python binding for SIDL long uses Python's unlimited precision integer data type, so it will not behave exactly like a 64 bit integer (i.e. there is no overflow). For Python versions before 2.2, your code needs to guarantee that a Python unlimited precision integer is used whenever a SIDL long is needed. For example, if you want to call a method whose SIDL signature is `bool isPrime(long num)`, calling `isPrime(1)` will fails; but calling `isPrime(1L)` will work fine.

The Python binding for an array of SIDL longs may use an array of 64 bit integers if Numeric Python supports a 64 bit integer. Otherwise, it uses an array of Python's indefinite precision integers (i.e., integers with unlimited bits).

What does this error message mean?

```
>>> import x.y.Zmodule
Traceback (innermost last):
File "<stdin>", line 1, in ?
ImportError:  dynamic module does not define init function (initZmodule)
```

Is the name of your SIDL interface/class `x.y.Z` or `x.y.Zmodule`, if it's the former, you should say **`import x.y.Z`**. If this isn't the problem, submit a bug report for Babel. It might be informative to look at the symbol of the shared library/dynamic link library using a tool like nm. I suppose it's also worth checking the PYTHONPATH environment variable to make sure it's pointing to the right place.

```
>>> import x.y.Z
Fatal Python error:  Cannot load implementation for SIDL class x.y.Z
Abort (core dumped)
```

This means that the Python stub code (the code that links Python to SIDL's independent object representation (IOR)) failed in its attempt to load the shared library or dynamic link library containing the implementation of SIDL class `x.y.Z`. Make sure the environment variable `SIDL_DLL_PATH` lists all the directories where the shared libraries/dynamic link libraries for your SIDL objects/interfaces are stored. `SIDL_DLL_PATH` is a semicolon separated list of directories where SIDL client stubs will search for shared libraries required for SIDL classes and interfaces. Make sure the directory in which the SIDL runtime resides is in the `LD_LIBRARY_PATH` (or whatever your machine's mechanism for locating shared library files is).

```
>>> import x.y.Z
Fatal Python error:  Cannot load implementation for SIDL interface x.y.Z
Abort (core dumped)
```

This is the same problem for an interface as described immediately above for a class.

11.8 How to Implement SIDL Objects in Python

To build server side Python, you must have Python compiled as a shared library or dynamically link library. The standard Python build only builds the necessary shared library on a few platforms — none of which are target platforms

for Babel. Some Linux distributions include a Python shared library, and it is possible to make a Python shared library on Solaris. The Python shared library should contain the objects from libpythonx.y.a where x.y is your Python version. Making a shared library is different on each platform, so it is not covered here.

To implement an object in Python, first you must run Babel to create the Python server side bindings [1].

```
% babel --exclude-external --server=python file.sidl
```

or simply

```
% babel -E -s=python file.sidl
```

This creates the IOR, Python skeleton (pSkel), and Python launch (pLaunch) files in your currect directory, and it will create tree of subdirectories based on the package hierarchy found in file.sidl. The IOR, pSkel and pLaunch files must be compiled and place in a shared library (in most cases).

The tree of subdirectories created by Babel includes Python implementation files whose name ends with `_Impl.py` and Python extension modules for the Python client side binding (`_Module.h` and `_Module.c`). The extension modules need to be compiled as above in section 11.5, and you need to fill in the implementations in the `_Impl.py` files.

Babel generates the outline of the implementation. It creates a class definition and empty methods for you to fill in the each `_Impl.py` file. If you put your code between the comments as indicated, your code will be preserved if you rerun Babel. Any changes out side the comment blocks will be lost if you rerun Babel. Here is an example of a method implementation:

Python

```
def passeverywhere(self, d1, d3):
  #
  # SIDL EXPECTED INCOMING TYPES
  # ============================
  # double d1
  # double d3
  #

  #
  # SIDL EXPECTED RETURN VALUE(s)
  # =============================
  # (_return, d2, d3)
  # double _return
  # double d2
  # double d3
  #

  # DO-NOT-DELETE splicer.begin(passeverywhere)
  if (d1 == 3.14):
    retval = 3.14
  else:
    retval = 0
  return (retval, 3.14, -d3)
  # DO-NOT-DELETE splicer.end(passeverywhere)
```

Babel generated everything except the code that appears between the `splicer.begin` and `splicer.end` comments.

[1] For information on additional command line options, refer to Section 3.2.

Chapter 12

SIDL Backend

Contents

12.1 Introduction

This chapter introduces the SIDL backend associated with symbols that may originate from a SIDL file or the corresponding Extensible Markup Language (XML) representation. Unlike most of the other supported language bindings, the output from this backend is textual in nature. That is, it is the textual, human-readable form of the interfaces description. An alternative text form, XML that is, which is also supported is described in Chapter 13.

12.2 Purpose

The primary reason for having a SIDL backend is to provide a mechanism for generating human-readable text for interfaces that are written in conformant XML. It is important to emphasize that Babel requires the XML to conform to the SIDL DTD in order to benefit from this feature.

Generating SIDL provides a feature to collaborators who are interested in experimenting with the XML form of the interfaces. Such groups should find the more human-readable descriptions of the interfaces to be helpful for distribution and discussion.

12.3 Generated versus Original SIDL files

Generated SIDL files may differ from their original SIDL files in several respects in terms of content as well as layout. These differences are summarized below.

Packages. The code generation is limited to one high-level package per generated file. In fact, the name of the generated file is currently defined to be the concatenation of the name of the highest-level package and `.sidl`.

Versioning. The generation of requires statements is inferred from the symbols that actually appear in the associated interface descriptions. The intent is to provide a requires statement for only the highest level package needed of a given version. Consequently, requires and imports statements that were not necessary for resolving symbols will not appear. Also, fully qualified names will be shortened in the generated files due to the automatic generation

of the associated requires statement(s). Finally, since an import and require statement can be used in a SIDL file and no distinction is made in the XML, only a require statement will appear in the generated file.

Implements. Since there is no distinction between `implements-all` and `implements` in the XML version of the interfaces, the generated code outputs `implements` along with the inherited methods.

Comments. Babel preserves only document, or doc, comments so any comments that do not conform will not appear in the generated file [1].

Whitespace. Obviously there may be whitespace differences in the generated file. These include indentation, blank spaces and lines, and brace placement.

As an example, suppose we have a package in the file `foo.sidl`. The original file's contents are:

SIDL
```
package foo version 1.0 {

  class A {}

  package bar version 2.0 {
    class B {}
  }

}
```

The resulting contents of the generated SIDL file are:

SIDL
```
package foo version 1.0 {

  class A {
  }

  package bar version 2.0 {

    class B {
    }

  }


}
```

Notice the differences in white space. To illustrate more features, further suppose we have a package in the file `fooTest.sidl`. The original file's contents are:

SIDL
```
// An ignored comment
require foo version 1.0;
require foo.bar version 2.0;

/**
 * Test of doc comment with XML special characters < & >.
 */
package fooTest version 0.1 {

  /**
   * Another doc comment for an empty class.
```

[1] For more information on comments and doc-comments, refer to **Comments and Doc-Comments** in Section 5.2.

```
   */
  class A extends foo.bar.B {}

  class B extends foo.A {}
}
```

The resulting contents of the generated SIDL file are:

SIDL

```
require foo version 1.0;
require foo.bar version 2.0;

/**
 * Test of doc comment with XML special characters < & >.
 */
package fooTest version 0.1 {

  /**
   * Another doc comment for an empty class.
   */
  class A extends foo.bar.B {
  }

  class B extends foo.A {
  }

}
```

Here we see the exclusion of non-document comments and the retention of document comments. Refer to Section 5.2 and Appendix C for more information about document comments.

12.4 XML File Comparison

Testing has revealed that XML generated from the original SIDL file compared to XML generated from generated SIDL files have only minor differences. In fact, the differences are limited to specific metadata fields. Specifically, the date, source-url, and source-line entries can differ. The dates, however, will be the same if the "`--suppress-timestamp`" option was used when both XML files were generated. Similarly, the source-line entries will be the same if the package started on the same line in both the original and generated SIDL files. The latter can happen if, for instance, there are no non-doc comments in the original file.

12.5 Babel Command Line Options

To generate SIDL from a file using the default repository to resolve symbols, you should invoke Babel as follows [2]:

```
% babel --exclude-external --text=SIDL file.sidl
```

or use the short form

```
% babel -E -tSIDL file.sidl
```

Alternatively, you can generate SIDL from XML symbols, again assuming the default repository is used to resolve symbols, by typing the following at the command line:

[2] For information on additional command line options, refer to Section 3.2.

```
% babel --exclude-external --text=SIDL packagename
```

or use the short form

```
% babel -E -tSIDL packagename
```

Chapter 13

XML Backend

Contents

13.1 Introduction

This chapter introduces the XML representation supported by Babel. Here we describe the motivation for having an XML backend and the basic structure of a conformant XML file. To illustrate, a few of the SIDL symbol XML files will be presented.

Details regarding the layout of XML files can be obtained by referring to the Document Type Definition (DTD) provided in Appendix C. For more on the type repositories, refer to[XML Repositories in Section 5.2.

13.2 Purpose

The XML backend is a key feature of Babel. It provides the basis upon which the symbol, or type, repository depends. SIDL files should be translated into their XML representations and stored in the type repository. This is the case for the SIDL interfaces and classes that are provided as part of the Babel toolkit.

13.3 Basic Structure

Each generated XML file specifies the interfaces for a given SIDL Symbol in an expanded textual representation. Although the structure of a given file depends upon the type of symbol it contains, the basic layout consists of a set of common elements followed by symbol-specific elements.

Common Elements

The common elements are *prolog*, *document type*, *name*, *metadata*, and *comment*. These elements, which are described below, are followed by symbol-specific information.

Prolog. The prolog simply identifies the XML version and encoding scheme associated with the file.

Document Type. The document type declaration states the document contains a *Symbol* and it identifies the associated DTD (i.e., *SIDL.dtd*).

Name. The symbol name is the first element within the symbol tag pair and it identifies the name and version of the SIDL symbol that is described in the file.

Metadata. The metadata element identifies the date the XML file was generated[1] along with a set of three key-value pair entries. The first, *source-url*, identifies the URL of the SIDL file that was used to generate the XML file. The second, *source-line*, identifies the line within the SIDL file at which the symbol was first detected. Finally, *babel-version* identifies the version of Babel that was used to generate the XML file.

Comment. The comment tag is used to save off any comment that is associated with the symbol.

Packages

In addition to the common elements, packages retain elements and attributes associated with SIDL packages. These include whether or not the package is *final* along with a list of the symbols contained within the package. The list of symbols consists of the tuple: name, type, and version.

For example, the XML representation of the toplevel SIDL package (i.e., `sidl`) is:

shell

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Symbol PUBLIC "-//CCA//sidl Symbol DTD v1.1//EN"
  "/babel/share/repository/sidl.dtd">
<Symbol>
  <SymbolName name="sidl" version="0.9.12" />
  <Metadata date="20051208 10:47:28 PST">
    <MetadataEntry key="source-url"
    value="file:/babel/runtime/sidl/sidl.sidl" />
    <MetadataEntry key="babel-version" value="0.10.51" />
    <MetadataEntry key="xml-url"
    value="/babel/share/repository/sidl-v0.9.12.xml" />
    <MetadataEntry key="source-line" value="39" />
  </Metadata>
  <Comment>The
  <code>sidl</code>package contains the fundamental type and
  interface definitions for the
  <code>sidl</code>interface definition language. It defines common
  run-time libraries and common base classes and interfaces. Every
  interface implicitly inherits from
  <code>sidl.BaseInterface</code>and every class implicitly
  inherits from
  <code>sidl.BaseClass</code>.</Comment>
  <Package final="false">
    <PackageSymbol name="BaseInterface" type="interface"
    version="0.9.12" />
    <PackageSymbol name="BaseClass" type="class"
    version="0.9.12" />
    <PackageSymbol name="io" type="package" version="0.9.12" />
    <PackageSymbol name="BaseException" type="interface"
    version="0.9.12" />
    <PackageSymbol name="RuntimeException" type="interface"
    version="0.9.12" />
    <PackageSymbol name="SIDLException" type="class"
    version="0.9.12" />
    <PackageSymbol name="PreViolation" type="class"
    version="0.9.12" />
    <PackageSymbol name="PostViolation" type="class"
```

[1] Assuming the "`--suppress-timestamp`" option was not used.

```
    version="0.9.12" />
    <PackageSymbol name="InvViolation" type="class"
    version="0.9.12" />
    <PackageSymbol name="Scope" type="enum" version="0.9.12" />
    <PackageSymbol name="Resolve" type="enum" version="0.9.12" />
    <PackageSymbol name="DLL" type="class" version="0.9.12" />
    <PackageSymbol name="Finder" type="interface"
    version="0.9.12" />
    <PackageSymbol name="DFinder" type="class" version="0.9.12" />
    <PackageSymbol name="Loader" type="class" version="0.9.12" />
    <PackageSymbol name="ClassInfo" type="interface"
    version="0.9.12" />
    <PackageSymbol name="ClassInfoI" type="class"
    version="0.9.12" />
    <PackageSymbol name="MemoryAllocationException" type="class"
    version="0.9.12" />
    <PackageSymbol name="CastException" type="class"
    version="0.9.12" />
    <PackageSymbol name="LangSpecificException" type="class"
    version="0.9.12" />
    <PackageSymbol name="rmi" type="package" version="0.9.12" />
  </Package>
</Symbol>
```

Interfaces

Similarly, the XML for interface symbols contain the common elements. In addition, they retain elements and attributes associated with SIDL interfaces. These include any extensions, parent interfaces it implements, and its methods. Method information includes its name, communication mode, short name, name extension (for languages that don't support method overloading), comment, return type, argument list, and exception list.

For example, the XML representation of *sidl.BaseInterface* is:

shell

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Symbol PUBLIC "-//CCA//sidl Symbol DTD v1.1//EN"
  "/babel/share/repository/sidl.dtd">
<Symbol>
  <SymbolName name="sidl.BaseInterface" version="0.9.12" />
  <Metadata date="20051208 10:47:28 PST">
    <MetadataEntry key="source-url"
    value="file:/babel/runtime/sidl/sidl.sidl" />
    <MetadataEntry key="babel-version" value="0.10.51" />
    <MetadataEntry key="xml-url"
    value="/babel/share/repository/sidl.BaseInterface-v0.9.12.xml" />
    <MetadataEntry key="source-line" value="46" />
  </Metadata>
  <Comment>Every interface in
  <code>sidl</code>implicitly inherits from
  <code>BaseInterface</code>, and it is implemented by
  <code>BaseClass</code>below.</Comment>
  <Interface>
    <ExtendsBlock />
    <AllParentInterfaces />
    <MethodsBlock>
      <Method communication="normal" copy="false"
```

```
definition="abstract" extension="" shortname="addRef">
  <Comment>
    <p>Add one to the intrinsic reference count in the
    underlying object. Object in
    <code>sidl</code>have an intrinsic reference count.
    Objects continue to exist as long as the reference count
    is positive. Clients should call this method whenever
    they create another ongoing reference to an object or
    interface.</p>
    <p>This does not have a return value because there is no
    language independent type that can refer to an interface
    or a class.</p>
  </Comment>
  <Type type="void" />
  <ArgumentList />
  <ThrowsList />
  <ImplicitThrowsList>
    <SymbolName name="sidl.RuntimeException"
    version="0.9.12" />
  </ImplicitThrowsList>
</Method>
<Method communication="normal" copy="false"
definition="abstract" extension="" shortname="deleteRef">
  <Comment>Decrease by one the intrinsic reference count in
  the underlying object, and delete the object if the
  reference is non-positive. Objects in
  <code>sidl</code>have an intrinsic reference count. Clients
  should call this method whenever they remove a reference to
  an object or interface.</Comment>
  <Type type="void" />
  <ArgumentList />
  <ThrowsList />
  <ImplicitThrowsList>
    <SymbolName name="sidl.RuntimeException"
    version="0.9.12" />
  </ImplicitThrowsList>
</Method>
<Method communication="normal" copy="false"
definition="abstract" extension="" shortname="isSame">
  <Comment>Return true if and only if
  <code>obj</code>refers to the same object as this
  object.</Comment>
  <Type type="boolean" />
  <ArgumentList>
    <Argument copy="false" mode="in" name="iobj">
      <Type type="symbol">
        <SymbolName name="sidl.BaseInterface"
        version="0.9.12" />
      </Type>
    </Argument>
  </ArgumentList>
  <ThrowsList />
  <ImplicitThrowsList>
    <SymbolName name="sidl.RuntimeException"
```

```
          version="0.9.12" />
        </ImplicitThrowsList>
      </Method>
      <Method communication="normal" copy="false"
      definition="abstract" extension="" shortname="isType">
        <Comment>Return whether this object is an instance of the
        specified type. The string name must be the
        <code>sidl</code>type name. This routine will return
        <code>true</code>if and only if a cast to the string type
        name would succeed.</Comment>
        <Type type="boolean" />
        <ArgumentList>
          <Argument copy="false" mode="in" name="name">
            <Type type="string" />
          </Argument>
        </ArgumentList>
        <ThrowsList />
        <ImplicitThrowsList>
          <SymbolName name="sidl.RuntimeException"
          version="0.9.12" />
        </ImplicitThrowsList>
      </Method>
      <Method communication="normal" copy="false"
      definition="abstract" extension="" shortname="getClassInfo">
        <Comment>Return the meta-data about the class implementing
        this interface.</Comment>
        <Type type="symbol">
          <SymbolName name="sidl.ClassInfo" version="0.9.12" />
        </Type>
        <ArgumentList />
        <ThrowsList />
        <ImplicitThrowsList>
          <SymbolName name="sidl.RuntimeException"
          version="0.9.12" />
        </ImplicitThrowsList>
      </Method>
    </MethodsBlock>
  </Interface>
</Symbol>
```

Classes

Class definitions are almost identical to that of interfaces except for additional attributes. The additional attribute, which include whether or not the class is *final*. Recall that Babel/SIDL supports only single inheritance of classes; therefore, only a single class will appear in the extends block. If one does not appear in the original SIDL file, by default the class will extend *sidl.Baseclase*.

For example, the XML representation of *sidl.BaseClass* is:

shell

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Symbol PUBLIC "-//CCA//sidl Symbol DTD v1.1//EN"
  "/babel/share/repository/sidl.dtd">
<Symbol>
  <SymbolName name="sidl.BaseClass" version="0.9.12" />
  <Metadata date="20051208 10:47:28 PST">
```

```
  <MetadataEntry key="source-url"
  value="file:/babel/runtime/sidl/sidl.sidl" />
  <MetadataEntry key="babel-version" value="0.10.51" />
  <MetadataEntry key="xml-url"
  value="/babel/share/repository/sidl.BaseClass-v0.9.12.xml" />
  <MetadataEntry key="source-line" value="97" />
</Metadata>
<Comment>Every class implicitly inherits from
<code>BaseClass</code>. This class implements the methods in
<code>BaseInterface</code>.</Comment>
<Class abstract="false">
  <Extends />
  <ImplementsBlock>
    <SymbolName name="sidl.BaseInterface" version="0.9.12" />
  </ImplementsBlock>
  <AllParentClasses />
  <AllParentInterfaces>
    <SymbolName name="sidl.BaseInterface" version="0.9.12" />
  </AllParentInterfaces>
  <MethodsBlock>
    <Method communication="normal" copy="false"
    definition="final" extension="" shortname="addRef">
      <Comment>
        <p>Add one to the intrinsic reference count in the
        underlying object. Object in
        <code>sidl</code>have an intrinsic reference count.
        Objects continue to exist as long as the reference count
        is positive. Clients should call this method whenever
        they create another ongoing reference to an object or
        interface.</p>
        <p>This does not have a return value because there is no
        language independent type that can refer to an interface
        or a class.</p>
      </Comment>
      <Type type="void" />
      <ArgumentList />
      <ThrowsList />
      <ImplicitThrowsList>
        <SymbolName name="sidl.RuntimeException"
        version="0.9.12" />
      </ImplicitThrowsList>
    </Method>
    <Method communication="normal" copy="false"
    definition="final" extension="" shortname="deleteRef">
      <Comment>Decrease by one the intrinsic reference count in
      the underlying object, and delete the object if the
      reference is non-positive. Objects in
      <code>sidl</code>have an intrinsic reference count. Clients
      should call this method whenever they remove a reference to
      an object or interface.</Comment>
      <Type type="void" />
      <ArgumentList />
      <ThrowsList />
      <ImplicitThrowsList>
```

```
      <SymbolName name="sidl.RuntimeException"
      version="0.9.12" />
    </ImplicitThrowsList>
  </Method>
  <Method communication="normal" copy="false"
  definition="final" extension="" shortname="isSame">
    <Comment>Return true if and only if
    <code>obj</code>refers to the same object as this
    object.</Comment>
    <Type type="boolean" />
    <ArgumentList>
      <Argument copy="false" mode="in" name="iobj">
        <Type type="symbol">
          <SymbolName name="sidl.BaseInterface"
          version="0.9.12" />
        </Type>
      </Argument>
    </ArgumentList>
    <ThrowsList />
    <ImplicitThrowsList>
      <SymbolName name="sidl.RuntimeException"
      version="0.9.12" />
    </ImplicitThrowsList>
  </Method>
  <Method communication="normal" copy="false"
  definition="normal" extension="" shortname="isType">
    <Comment>Return whether this object is an instance of the
    specified type. The string name must be the
    <code>sidl</code>type name. This routine will return
    <code>true</code>if and only if a cast to the string type
    name would succeed.</Comment>
    <Type type="boolean" />
    <ArgumentList>
      <Argument copy="false" mode="in" name="name">
        <Type type="string" />
      </Argument>
    </ArgumentList>
    <ThrowsList />
    <ImplicitThrowsList>
      <SymbolName name="sidl.RuntimeException"
      version="0.9.12" />
    </ImplicitThrowsList>
  </Method>
  <Method communication="normal" copy="false"
  definition="final" extension="" shortname="getClassInfo">
    <Comment>Return the meta-data about the class implementing
    this interface.</Comment>
    <Type type="symbol">
      <SymbolName name="sidl.ClassInfo" version="0.9.12" />
    </Type>
    <ArgumentList />
    <ThrowsList />
    <ImplicitThrowsList>
      <SymbolName name="sidl.RuntimeException"
```

```
          version="0.9.12" />
        </ImplicitThrowsList>
      </Method>
    </MethodsBlock>
  </Class>
</Symbol>
```

13.4 Command Line Options

XML must be generated from a SIDL file. The Babel command line is as follows [2]:

```
% babel --exclude-external --text=XML file.sidl
```

or simply

```
% babel -E -tXML file.sidl
```

In both cases, the use of the default repository is assumed for resolving symbols. In addition, the output will appear in the default output directory.

[2] For information on additional command line options, refer to Section 3.2.

Chapter 14

HTML Interface Documentation

Contents

14.1 Introduction

Babel can automatically create interface documentation using the HTML backend. This capability is modeled after the javadoc documentation available with Java. It is invoked with the `--text=html` command line option.

Part III

Advanced Topics

Chapter 15

Remote Method Invocation

Contents

15.1 What is RMI?

In classic Babel, all calls are in-process. That is, everything Babel generates is loaded into the same process. This means inter-language Babel calls use the same mechanisms as normal function calls. This makes calls between languages extremely fast. However, many systems also have a need for Remote Procedure Calls (RPC), that is, calls made between different processes, or even machines over a network. Remote Method Invocation (RMI) is Babel's answer to this need.

RMI is Object Oriented RPC. However, unlike RPC where calls are made to procedures on a specified machine, in RMI calls are made on objects. The call is run on whichever machine the object resides on.

There are several reasons why an application may choose to use RMI. The main reasons are wrapping code tied to particular hardware, wrapping code tied to a particular operating system release, coarse-grained parallel execution, or greater encapsulation. With RMI, you can make code that's tied to a particular machine available to programs running on other platforms. You can utilize multi-CPU systems to concurrently solve problems using RMI. RMI can solve

problems that sometimes occur when you put two codes in the same address space. For example, two Fortran codes may use the same logical unit numbers (similar to C file numbers), or two codes may both need a customized form of a third party library. Bringing both codes into the same process may can a symbol collision for the third party library, and one code gets the wrong version of the library.

Despite the radical low-level differences between RMI and classic Babel, the user interfaces are nearly identical. In fact, if a library writer does not care if an object is remote or not, they simply do not need to know. RMI support requires a few simple calls to set up the infrastructure, but almost everything else is handled automatically by the Babel runtime library. Babel also has a few RMI support functions and a special remote constructor.

15.2 Babel RMI Concepts

For normal function calls, arguments are passed by initializing registers and pushing things onto the system stack. In RMI, function arguments and return values are passed by a network protocal. From a programmers point of view, the only difference between normal and network function calls is that network function calls have more failure modes. Anything that can disturb a network connection, such as a router going offline, can cause a RMI call to fail.

Conceptually, the RMI view of the world can be thought of as 1 or more Babel Object Servers (BOSs) that a client can connect to in order to create or use objects on those servers. Of course, any server can also connect as a client to any other server, and any client can become a server simply by starting up a BOS of it's own.

This makes Babel RMI very flexible, and accepting of whatever client-server relationships the application writers choose to use. Web Services users of Babel tend to use traditional client-server models, while scientific distributed systems users tend toward peer-to-peer usage.

15.2.1 RMI Protocols

The first thing any user of Babel RMI has to do is choose a Babel RMI protocol. Babel RMI can use any protocol that implements the Babel RMI API, but a client and a server using different protocols probably cannot communicate.

A Babel RMI protocol may be built on any low level protocol (such as TCP/IP) that the protocol implementor wishes to use. This should not affect the user at all. The protocol controls the details of how arguments and return values are converted to a stream of bytes that can be shipped across the network and read by the other process.

Currently there is only one protocol that fully implements the Babel RMI API, and it is included with Babel in the runtime/sidlx directory. It is called "Simple Protocol." However, there are many other protocols currently under development (at least four.) Soon there will be a number of choices, including protocols specifically tuned for high-performance scientific computing, web services, and CORBA[17] compatibility.

15.2.2 Babel Object Server

The next thing that a user needs is a BOS to connect to. The BOS is implemented by the protocol writers as a library. As mentioned before, the BOS may be run by itself by a small driver program, or run as part of a program that also acts as a client. There is an example of a small driver program in contrib/babel-rmi/orb.

The BOS is accessed by a protocol specific URL. A URL is a string the uniquely identifies a network resource. Most people are aware of Internet URLs like: `http://www.llnl.gov/CASC/components/babel.html` (which is the URL for the Babel web page). Babel RMI also uses URLs, but they are mostly protocol specific. Babel RMI only uses the portion of the URL up to the first non-alphanumeric character to identify the protocol that is being used. The rest of the URL is passed on to the protocol. This means that while "Simple Protocol" URLs look like this:

`simhandle://pc3.nowhere.com:9999/`

one can also imagine URLs of the form:

`weird://05:16:5B:BD:E1:73/`

for the weird protocol, or:

`weird+SSL://05:16:5B:BD:E1:73/`

for running the weird protocol over SSL. Babel RMI itself does not attempt to parse anything past the first non-alphanumeric character, so most of the URL is entirely protocol dependent.

15.2.3 Object Creation and Connection

There are two main ways of accessing objects on a remote server, creation and connection.

A client may create a remote object with the Babel built-in *static _createRemote(in string URL)* method. This asks the remote server given in the URL to create an object. For example, the C function

`foo_Bar b = foo_Bar__createRemote(''simhandle://pc3:9999/'');`

will create an object of type foo.Bar on the server running on pc3 port 9999 using the "Simple Protocol."

`foo_Bar b` may now be passed around exactly like a normal Babel object, except that all calls on b will actually run on pc3.

However, in some cases, an object already exists on a remote server that the user just wants to access. In this case, the object can be connected to via the built-in *static _connect(in string URL)* method. The only difference is, in this case the URL must include an object ID to uniquely identify the object desired on the BOS. For example:

`foo_Bar b = foo_Bar__connect(''simhandle://pc3:9999/Bar1025'');`

here again, `foo_Bar b` may now be used exactly like any other Babel object. To relate back to normal Babel, connection is kind of like an active addRef. The user actively goes and gets his own reference to a given object.

`_connect` is actually used by Babel internally whenever objects are passed remotely as arguments. In fact, users will probably rarely use connect directly, most often it will be done automatically by Babel when objects are passed remotely. `_connect` is exposed to the user mostly for Web Services, where the objectID may always be the same, and for special boot strapping uses.

15.2.4 RMI Arguments

All basic types are passed by value in Babel RMI. They are actually copied across the network. This is reasonable since they are small. Arrays are also copy-only, so anytime an array is passed remotely through Babel RMI, it is actually copied to the remote machine.

Because arrays are copy-only for RMI, there is a noticable difference in the behavior of an `in array` argument between a non-RMI call and an RMI call. For the non-RMI call, the code implementing the method can change elements of the incoming array. Because the caller and callee share the same array, the caller's copy of the array will also be changed. For an RMI call, even if the code implementing the method changes elements of the incoming array, the caller's copy can never be modified because the client and server each have a distinct copy of the array data.

There are two ways one can pass objects in Babel RMI, by reference, and by copy. The default method is pass-by-reference. For example, server A calls a function foo on server B, and passes a local object Bar as an argument. In this case A will actually pass the URL of Bar to B, B will then call `_connect` on the URL, which connects back to the object Bar on A.

Pass-by-copy (also called serialization) is different. Pass-by-copy means that a new object is actually created locally on B, and filled in with the values from the object Bar on A. The result is two distinct local objects, one on A and one on B. In order to pass by copy, copy must be used as an argument modifier in the SIDL file. For example:

copy Bar retBar(copy in Foo f)

This sidl function takes a copy of a Foo and returns a copy of a Bar.

Passing by copy also requires the the object being passed implements *sidl.io.Serializable*:

SIDL

```
1 package sidl.io version 9.15 {
2   interface Serializable {
3     void packObj( in Serializer ser );
4     void unpackObj( in Deserializer des );
5   }
6 }
```

Serializable declares two methods, packObj and unPackObj. packObj serializes the internal object data to a string. unPackObj reinstates the data into the new object by unserializing it from a string. The library developer must implement these functions because Babel does not know what data is in the object, or how it should be serialized. Examples of packObj and unpackObj implementations can be found in *sidl.rmi.SIDLException* and *sidl.rmi.NetworkException*.

15.2.5 Casting Remote Objects

Babel RMI casting works the same as normal Babel casting, the user calls casts an object to a new type, and gets a new reference back of the object of the new type. In normal Babel, the new reference points to the same IOR object as the old reference. This is because all Babel objects are internally represented as the type they were created as, so casting is simply a matter of checking if the internal Babel type extends the target type or not.

Babel RMI objects are more complex, a cast may result in a new stub. If `_connect()` is called on a remote object, the object can be connected as a super type of its actual type, such as an interface. If this object is later cast to a more derived type, a new local object stub must be created. These two stubs must be deleteRef'd individually.

Here is an example where foo_Quux extends foo_Bar. The first is what the user should do, the second is an error.

```
foo_Bar fb = foo_Bar__connect("simhandle://pc1:9999/quux1234", &_ex);
foo_Quux fz = foo_Quux__cast(fb);
foo_Bar_deleteRef(fb);
foo_Quux_deleteRef(fz); //object is destroyed
```
ANSI C

Do not do this:

```
foo_Bar fb = foo_Bar__connect("simhandle://pc1:9999/quux1234", &_ex);
foo_Quux fz = foo_Quux__cast(fb);
foo_Bar_deleteRef(fb);
foo_Quux_deleteRef(fb); //ERROR!!!
```
ANSI C

15.3 Babel RMI Usage

The previous section generally covered the capabilities of RMI. This section covers the actual usage of those features.

15.3.1 Adding Protocols

In a normal Babel RMI client program, the first thing that needs to be done is adding any protocols that the user plans to use to the ProtocolFactory. The ProtocolFactory is a mapping from protocol prefix that normally proceeds a URL, and the protocol's actual implementation. The only method the user ever needs to access is `addProtocol`.

```
static bool addProtocol( in string prefix, in string typeName );
```

`addProtocol` takes the protocol prefix and the fully qualified SIDL protocol typename. It returns `TRUE` on success. For example, normally the shortname for the "Simple Protocol" protocol is "simhandle." So we would call the ProtocolFactory like this:

```
sidl_rmi__addProtocol(``simhandle'',''sidl.rmi.SimHandle'');
```

Now Babel RMI knows what to call when it encounters a URL prefixed by "simhandle://".

15.3.2 Built-in Functions

We already covered two important built-in Babel RMI functions in Section 15.2.3. They were the `_create[Remote]` and `_connect` static built-in functions, which remotely create an object or connect to an already existing remote object respectively.

There are three other important RMI related built-in functions that a user may find useful. The first two are related:

```
bool _isRemote();
bool _isLocal();
```

`_isLocal()` and `_isRemote()` are opposites. `_isRemote()` returns `TRUE` if the object it is called on is a remote object. `_isLocal()`, on the other hand, returns `TRUE` if the object is implemented locally.

Many Babel RMI users will never need these functions. If you don't care where an object exists, or you already know statically, these methods are totally superfluous. However, since calls on remote objects have serious performance implications, we included these functions for convenience.

There is one other important RMI related built-in:

```
string _getURL();
```

This function returns the URL of the object it is called on. This is straight forward for Remote objects, but for local objects it may have some interesting side effects. First, if there is no BOS running locally, no local objects can be exported remotely. Therefore no local object will have a URL. In this case `_getURL()` returns `NULL`. However, if there is a BOS running, then local object may have a URL. In this case, if the object has already been exported, `_getURL()` will return the URL in the *sidl.rmi.InstanceRegistry*, if the object is not in the *InstanceRegistry*, Babel will add it, thereby automatically generating a URL for the object. To reiterate, a possible side-effect of calling `_getURL()` is that the object it is called on my be added to the *InstanceRegistry*.

15.3.3 Passing Objects from a client

If there is no BOS running locally, (that is, your process is strictly a client) you cannot expose your local objects to remote machines by reference. However, that doesn't mean you can't pass objects around. The client can still pass references to remote objects on other remote servers, and can still pass both local and remote objects by copy. For this section, we will take our examples from this SIDL:

SIDL
```
package foo version 0.1 {

  class Bar {
    void setBaz(in foo.Baz bz);
    void setBazCopy(in copy foo.Baz bz);
    //returns the registered Baz, or a new one if none exists
    foo.Baz returnBaz();
  }

  class Baz {}
}
```

From the above SIDL, you can see that the following C code is perfectly legal for a client:

ANSI C
```
foo_Bar fb = foo_Bar__createRemote("simhandle://pc1:9999", &_ex);
foo_Baz fz = foo_Bar_returnBaz(fb, &_ex);
foo_Baz_runSimulation(fz, &_ex);
```

It's legal because the remote call returns a reference to another remote object, the client never actually exports any of it's local objects.

The following chunk is also legal, because it passes a remote object to a different remote server. (Passing it to the same remote server would be OK too.)

ANSI C
```
foo_Bar fb = foo_Bar__createRemote("simhandle://pc1:9999", &_ex);
foo_Baz fz = foo_Baz__createRemote("simhandle://pc2:9999", &_ex);
foo_Bar_setBaz(fb, fz, &_ex);
```

And the following is ALSO legal, because clients can pass local objects remotely by copy, they just can't pass local objects by reference. (This allows users to drive a remote simulation on a cluster from a regular workstation with nothing but a simple client.)

ANSI C
```
foo_Bar fb = foo_Bar__createRemote("simhandle://pc1:9999", &_ex);
foo_Baz fz = foo_Baz__create(&_ex);  //Local object
foo_Bar_setBazCopy(fb, fz, &_ex);    //Pass by copy
```

However this final bit of code will throw an exception if run by a client that has no BOS:

ANSI C
```
/* X ILLEGAL X WILL THROW EXCEPTION X     */

foo_Bar fb = foo_Bar__createRemote("simhandle://pc1:9999", &_ex);
foo_Baz fz = foo_Baz__create(&_ex); //Local object
```

```
foo_Bar_setBaz(fb, fz, &_ex);        //Pass by reference X BAD! X

/* X ILLEGAL X WILL THROW EXCEPTION X      */
```

15.4 Babel Object Servers

Now that we've seen how to use a client, we will take a look at running a Babel Object Server.

15.4.1 Starting up a Babel Object Server

Babel Object Servers are generally easy to start up, although each BOS may have a different construction interface. Here is an example of starting up the "Simple Protocol"

ANSI C

```
sidlx_rmi_SimpleOrb echo=NULL;
char* url = "simhandle://localhost:9999";
int tid;
sidl_rmi_ServerInfo si = NULL;

echo = sidlx_rmi_SimpleOrb__create(&ex);SIDL_CHECK(ex);
sidlx_rmi_SimpleOrb_init( echo, url, 1, &ex);SIDL_CHECK(ex);
tid = sidlx_rmi_SimpleOrb_run( echo, &ex );SIDL_CHECK(ex);
si = sidl_rmi_ServerInfo__cast(echo,&ex);SIDL_CHECK(ex);
sidl_rmi_ServerRegistry_registerServer(si, &ex);SIDL_CHECK(ex);
sidl_rmi_ServerInfo_deleteRef(si,&ex);SIDL_CHECK(ex);

pthread_join(tid, NULL); //Optional PTHREAD join
```

Notice that before the server is run, `init` must be called. `init` takes two arguments: a URL and a set of flags. In "Simple Protocol" the URL is simply a method of communicating what port the server should listen to for connections. The port given may be a single port, or a range of ports. The rest of the URL is simply a formality. The flags variable is treated as a boolean that turns on verbosity if `TRUE`. Additional flags could be added in the future for more special features.

`run` returns a long. This return argument is meant to hold the thread id of the thread waiting for connections. The user may wish to join on the thread in order to keep the "Simple Protocol" server from exiting prematurely. (We are now past the "Simple Protocol" specific portion of this section)

After calling `run` the server is running, but you won't be able to export any local objects until you register the server with the *sidl.rmi.ServerRegistry*. Every BOS must be registered with the *ServerRegistry*, and therefore every BOS must implement the *sidl.rmi.ServerInfo* interface. This interface is what allows the server to interact with the *ServerRegistry*.

SIDL

```
class ServerRegistry {
  static void registerServer(in sidl.rmi.ServerInfo si);
  static string getServerURL(in string objID);
  static string isLocalObject(in string url);
  static array<sidl.io.Serializable,1> getExceptions();
}
```

The *ServerRegistry* is a singleton class that Babel RMI uses internally to interface with the BOS. It interfaces through the *sidl.rmi.ServerInfo* interface:

SIDL

```
interface ServerInfo {
  string getServerURL(in string objID);
  string isLocalObject(in string url);
  array<sidl.io.Serializable,1> getExceptions();
}
```

Simply cast the BOS to a *ServerInfo* and register it with the *ServerRegistry*.

The user is never really meant to use the *ServerInfo* interface. In some cases a user may wish to call `getExceptions()` through the *ServerRegistry*. `getExceptions()` is an advanced function. Usually, if there is an exception is raised in the BOS by a remote call, the exception is returned back to the caller. However, in some cases this is not possible. In those cases the BOS logs the exceptions. Later, a user may use getExceptions to get the logged exceptions.

NOTE: Currently the *ServerRegistry* can only handle one *ServerInfo*. This means that Babel can effectively only support one BOS at a time for exporting local objects. (There are hairy ways around this) This is because there are a lot of issues that appear when a user can export objects with a number of different protocols that we have not dealt with. This may be researched further in the future.

15.4.2 Publishing Babel Objects

Once you have a BOS up and running, you are free to export your local object to remote servers. (And, depending on your BOS, remote clients may be able to create and access objects on your BOS.) Exporting an object basically means that remote Babel processes can access the object. In implementation, this means that the object is accessible through the *sidl.rmi.InstanceRegistry*. The *InstanceRegistry* maps objectIDs to objects, and vice-versa. This is what allows a remote client to get a handle to your object with nothing more than a URL.

There are 3 ways to for an object to end up in the *InstanceRegistry*. The first, and easiest, is simply to pass a local object by-reference as an argument in a remote call. The last example in 15.3.3 works if a BOS is running.

ANSI C

```
/* THIS WORKS IF A BOS IS RUNNING   */
foo_Bar fb = foo_Bar__createRemote("simhandle://pc1:9999", &_ex);
foo_Baz fz = foo_Baz__create(&_ex); //Local object
foo_Bar_setBaz(fb, fz, &_ex);       //BOS is running, OK!
```

Another possibility is simply to call `_getURL()` on the local object when a BOS is running. This will add the object to the *InstanceRegistry*, so theoretically a remote client could access it. Although realistically the remote client would have to get the URL somehow.

The third possibility is to add it to the *InstanceRegistry* your self. The *InstanceRegistry* class:

SIDL

```
class InstanceRegistry {
  static string registerInstance( in sidl.BaseClass instance );
  static string registerInstance[ByString]( in sidl.BaseClass instance,
                                            in string instanceID);
  static sidl.BaseClass getInstance[ByString]( in string instanceID );
  static string getInstance[ByClass]( in sidl.BaseClass instance );
  static sidl.BaseClass removeInstance[ByString]( in string instanceID );
  static string removeInstance[ByClass]( in sidl.BaseClass instance );
}
```

calling *registerInstance* by itself results in the same thing as calling `_getURL` on the object, it puts the object in the registry, and returns a unique objectID. However, by calling *registerInstance[ByString]*, the user can supply their own objectID. This is useful for WebServices and bootstrapping. It is possible to explicitly publish and object with a special name. In fact, the InstanceRegistry allows aliasing, the same object can be in the registry multiple times under the same name.

However, there is one issue with using *registerInstance[ByString]*. What is there is already an object in the registry with that name? There are two possible cases, if the object under that name is the same object you are trying to register, the call is idempotent, it does nothing. However, if a different object in the registry already has that name, the *InstanceRegistry* registers the new object under a similar, but unique name. Usually a combination of the instanceID passed in by the user and a unique integer. This is usually the correct thing to do, but if the user really wants the object under the original name, they must call `removeInstance[ByString]` on the object that currently has that name, and re-register the new object.

NOTE: The *InstanceRegistry* does not addRef objects when they are inserted into it. You must not destroy an object you wish to be accessible remotely. This means that if you create an object, insert it into the instanceRegistry, and then deleteRef it, it will be destroyed. You must keep a reference to it until you wish to remove it from

the InstanceRegistry. (The InstanceRegistry does, however, addRef an objects that are gotten from it. If you call `getInstance[ByString]`, you will get a reference to that object and are free to deleteRef it.)

15.4.3 De-publishing Babel Objects

There are two ways to remove an object from the `InstanceRegistry`. The first, and most automatic, is for it's reference count to reach 0. When an object is destroyed it is automatically removed from the `InstanceRegistry`.

The other way to remove an object is to call `removeInstance[ByString]` or `removeInstance[ByClass]`. These will remove the objects from the registry without destroying them. They do not addRef however. So, if you create an object, insert it into the `InstanceRegistry`, remove it from the `InstanceRegistry` and then deleteRef it, it will be destroyed. (Assuming no one else as addRef'd it in the meantime.)

15.5 Non-Blocking Babel RMI

Non-Blocking RMI is an even more advanced topic, but it is essential to high-speed distributed computing. Non-Blocking RMI allows the user to mix work and communication. Many scientific computing related methods may take a very long time to complete, and the client might like to do some work while waiting. Non-blocking calls return immediately after sending the information to the server. When the response comes back, the user can make a special call to access the data. During the time between the send and the receive, the client is free to do other work.

There are two types of Non-blocking RMI in Babel, Nonblocking and oneway. Both are declared as attributes on the method in SIDL. The difference is that with oneway communication, the client does not expect any return values. A oneway method will not even return an exception, unless it occurs during communication with the server. On the other hand, a non-blocking call can have return arguments. The user will send a request, and get a `sidl.rmi.Ticket`. Later, the user may use the `Ticket` to receive the out arguments.

15.5.1 Protocols

Currently there are no protocols that actually support non-blocking RMI. "Simple Protocol" emulates support, as all protocols must, but the benefits are lost. However, there are at least two protocols currently under development that do support it.

15.5.2 Nonblocking SIDL

The SIDL declaring calls to be nonblocking and/or oneway:

SIDL
```
package foo version 0.2 {
  class Bar {
    nonblocking double runSimulation(in double x, inout y, out z);
    oneway void initSimulation(in string name, in int flags);
  }
}
```

Notice that the nonblocking call may take any arguments, but only in arguments are allowed for the oneway call.

NOTE: As of Babel 0.11.0, calling a non-blocking function on a local object causes a segfault. It works for remote objects, but not for local objects. Be careful. (Oneway calls are OK though)

15.5.3 Tickets

As mentioned previously, non-blocking RMI uses the class `sidl.rmi.Ticket` to handle the return values of non-blocking methods. There are actually two interfaces implemented by the Protocol that are used. `sidl.rmi.Ticket` and `sidl.rmi.TicketBook`

```
interface Ticket {
  void block();
  bool test();
  TicketBook createEmptyTicketBook();
  Response getResponse();  //For internal Babel use
}

interface TicketBook extends Ticket {
  void insertWithID( in Ticket t, in int id );
  int insert( in Ticket t );
  int removeReady( out Ticket t );
  bool isEmpty();
}
```
SIDL

sidl.rmi.TicketBook is, obviously, a collection of *Tickets*. A Ticket represents the out arguments of a single non-blocking call. The user may `test()` if the call has returned yet, or `block()` until it does. The user can also get an empty *TicketBook*.

The *TicketBook* is a little more complex. It extends *Ticket* as well as creating some of it's own functions. It is mostly just to allow a user to make a large amount of nonblocking calls and work while they return. This is a common design paradigm in highly parallel scientific computing. In the case of *TicketBook*, it is assumed the user will input a bunch of Tickets with IDs. Then he can either `block()` on all of them (waitall), `test()` to see if any have returned, or block on `removeReady` (waitany). `removeReady` will return the id that the *Ticket* was inserted with so that the user may identify it. Perhaps with a case statement.

One odd thing about *TicketBook* is that you can insert multiple tickets with the same name. *TicketBook* will not warn you or throw an exception if you double up on the same name. If two different Tickets are put in the TicketBook with the same name, there is guarantee about what order they will come out in, even if you remove by name.

15.5.4 Non-blocking Usage

The examples in this section will be written in C using the SIDL file given in Section 15.5.2

Calling a oneway Babel RMI function is syntactically exactly like calling a normal Babel function. The difference is just the danger of not being able to receive any exceptions beyond the initial communication. Example:

```
foo_Bar b1 = foo_Bar__createRemote("simhandle://pc1:9999", &_ex);SIDL_CHECK(_ex);

foo_Bar_initSimulation(b1, "Test Simulation 1", 0, &_ex); SIDL_CHECK(_ex);
```
ANSI C

Non-blocking calls are a bit more complex, requiring *Tickets* in order to get the return values. Here's an example program, now using a non-blocking call. Notice that the inout argument y is passed as an in argument in the send (as a value), and an out argument in the recv (as a pointer).

```
foo_Bar b1 = foo_Bar__createRemote("simhandle://pc1:9999", &_ex);SIDL_CHECK(_ex);
sidl_rmi_Ticket t = NULL;
double x, y, z;

foo_Bar_initSimulation(b1, "Test Simulation 1", 0, &_ex); SIDL_CHECK(_ex);
t = foo_Bar_runSimulation_send(b1, x, y, &_ex); SIDL_CHECK(_ex);
/* ... Work ... */
foo_Bar_runSimulation_recv(b, t, &y, &z, &_ex); SIDL_CHECK(_ex); //blocks on return
sidl_rmi_Ticket_deleteRef(t , &_ex); SIDL_CHECK(_ex);
```
ANSI C

Now, next is an example of a more complex program, that utilizes the power of *TicketBooks* to make multiple remote calls, work, and deal with the responses when they return.

ANSI C

```
foo_Bar b1 = foo_Bar__createRemote("simhandle://pc1:9999", &_ex);SIDL_CHECK(_ex);
foo_Bar b2 = foo_Bar__createRemote("simhandle://pc2:9999", &_ex);SIDL_CHECK(_ex);
foo_Bar b3 = foo_Bar__createRemote("simhandle://pc3:9999", &_ex);SIDL_CHECK(_ex);

sidl_rmi_Ticket t = NULL;
sidl_rmi_TicketBook tb = NULL;
double x, y, z;
int id1, id2, id3, tmpid;

foo_Bar_initSimulation(b1, "Test Simulation 1", 0, &_ex); SIDL_CHECK(_ex);
foo_Bar_initSimulation(b2, "Test Simulation 2", 0, &_ex); SIDL_CHECK(_ex);
foo_Bar_initSimulation(b3, "Test Simulation 3", 0, &_ex); SIDL_CHECK(_ex);

t = foo_Bar_runSimulation_send(b1, x, y, &_ex); SIDL_CHECK(_ex);
tb = sidl_rmi_Ticket_createEmptyTicketBook(t, &_ex); SIDL_CHECK(_ex);
id1 = sidl_rmi_TicketBook_insert(tb, t, &_ex); SIDL_CHECK(_ex);
sidl_rmi_Ticket_deleteRef(t , &_ex); SIDL_CHECK(_ex);

t = foo_Bar_runSimulation_send(b2, x, y, &_ex); SIDL_CHECK(_ex);
id2 = sidl_rmi_TicketBook_insert(tb, t, &_ex); SIDL_CHECK(_ex);
sidl_rmi_Ticket_deleteRef(t , &_ex); SIDL_CHECK(_ex);

t = foo_Bar_runSimulation_send(b3, x, y, &_ex); SIDL_CHECK(_ex);
id3 = sidl_rmi_TicketBook_insert(tb, t, &_ex); SIDL_CHECK(_ex);
sidl_rmi_Ticket_deleteRef(t , &_ex); SIDL_CHECK(_ex);

/* ... Work ... */

while(!sidl_tmi_TicketBook_isEmpty(tb, &_ex)) {
  SIDL_CHECK(_ex);
  tmpid = sidl_tmi_TicketBook_removeReady(&t,&_ex); SIDL_CHECK(_ex);
  switch(tmpid) {
  case id1:
    foo_Bar_runSimulation_recv(b, t, &y, &z, &_ex); SIDL_CHECK(_ex);
    /* Do something with data from Simulation 1 */
    break;
  case id2:
    foo_Bar_runSimulation_recv(b, t, &y, &z, &_ex); SIDL_CHECK(_ex);
    /* Do something with data from Simulation 2 */
    break;
  case id3:
    foo_Bar_runSimulation_recv(b, t, &y, &z, &_ex); SIDL_CHECK(_ex);
    /* Do something with data from Simulation 3 */
    break;

  }
  sidl_rmi_Ticket_deleteRef(t , &_ex); SIDL_CHECK(_ex);
}
```

Chapter 16

Building Portable Polyglot Software

Babel generates very portable source code for multilingual programing. There is also an art and science to transforming the source code to binary assets without breaking the language encapsulation Babel is trying to create. This chapter discusses the details: from the mundane issues of file layout, to the arcana of linker and loader flags.

Contents

16.1 Layout of Generated Files

Babel generates a lot of files. Many of these files you never have to look at in an editor, but they must all be compiled and properly linked into an application (see Section 16.2). In this section we discuss a host of flags that can affect where files get generated.

- `--output-directory`=*path*
  This sets the root directory of where your files will be generated. The path can be absolute, or relative to the current working directory.

- `--generate-subdirs`
  This option forces files to be laid out in a directory hierarchy following the package hierarchy in the SIDL file. This arrangement is required for the Java and Python languages, so those generators force this option on and allow no means to turn it off. For C/C++ and Fortran 77/90, the default is that all files be generated in the single output directory with no package-named subdirectories.

- `--language-subdir`
  This option was contributed by a user. This option appends a language-specific subdirectory (e.g. c, python, f77) to the end of the path.

- **--hide-glue**
  This option was contributed by a user. The intent here is to separate the Impl files (which must be modified) from all other files. If this flag is set, then wherever an Impl file gets generated, all the corresponding Skels, Stubs, IORs, etc get generated in a subdirectory named `glue`.

Arbitrary combinations of the above flags are allowed. Regardless of the order they appear in the commandline, they are applied to the resulting path in the order they are presented above. For example if a SIDL file `pkg.sidl` defines a *`Cls`* class in the *`pkg`* package, and the user runs Babel as follows:

```
% babel -lugo there -sc
```

Then the majority of the sources will be generated in the `there/pkg/c/glue/` directory (except the Impl files which will occur one directory up in `there/pkg/c/`). Note the use of equivalent short-form commands in this example. If readers wish to review long and short forms of command line arguments, see Tabel 3.1 on page 16.

Note that many of these options were contributed by users and are not employed in Babel's own build. Instead, we tend to put a SIDL file in a directory and then generate client-side or server-side bindings in in either `runXXX/` or `libXXX/` subdirectories, respectively (where `XXX` is a language name). We don't use the **--generate-subdirs** or **--hide-glue** flags because they place source files that belong in the same library in different directories. Automake, which Babel uses as part of its build system, works much more reliably when all the sources that go into a library appear in the same directory as the library to be. The **--language-subdir** has a similar effect to what we do manually, but doesn't capture if it was client-side or server-side. In our tests and demos, we tend to build these separately because we want to exercise different drivers with different implementations.

16.2 Grouping compiled assets into Libraries

Babel enables one to completely encapsulate language dependencies inside a static or dynamically loaded library. This means that one can take a SIDL file and a compiled library, generate the bindings they want in their language of choice from the SIDL file, link against the library, and use it... never knowing what the original implementation language is for the library.

Babel generates the source code to accomplish this level of language interoperability, but users must use their compilers and linkers correctly for the effect to be complete. This section deals with many of the details that

16.2.1 Basics of Compilation and Linkage

What we generally think of as a compiler is really an ensemble of related tools. Generally there is a preprocessing step where very simple transformations occur (e.g. #define, #include directives and others). Next, the compiler proper executes and typically transforms your sourcecode into assembler or some other intermediate form. Optimizers work on this intermediate form and do perform additional transformations. Most big vendors of C, C++, and Fortran compilers have a common optimizer for all languages. Next, assemblers transform the optimized codes into platform-specific binaries. But this is not the end. The binaries may be linked together into libraries or programs. Libraries can be linked against other libraries, and eventually multiple programs. The main difference is that a program has additional instructions to bootstrap itself, do some interaction with the operating system, receive an argument list, and call `main()`. To see all this in action, try building a "hello world" type program in your favorite language, and run the "compiler" with an additional flag such as **-v**, **--verbose**, or whatever.

For example, this is what I get from a g77 compiler.

```
% g77 hello_world.f
% ./a.out
Hello World!  % g77 -v hello_world.f
Driving:  g77 -v hello_world.f -lfrtbegin -lg2c -lm -shared-libgcc
Reading specs from /usr/local/gcc/3.2/lib/gcc-lib/i686-pc-linux-gnu/3.2/specs
Configured with:  ../gcc-3.2/configure --prefix=/usr/local/gcc/3.2
Thread model:  posix
gcc version 3.2
```

```
/usr/local/gcc/3.2/lib/gcc-lib/i686-pc-linux-gnu/3.2/f771 hello_world.f
-quiet -dumpbase hello_world.f -version -o /tmp/ccp2GBGE.s
GNU F77 version 3.2 (i686-pc-linux-gnu)
compiled by GNU C version 3.2.
as --traditional-format -V -Qy -o /tmp/ccEiIsHc.o /tmp/ccp2GBGE.s
GNU assembler version 2.11.90.0.8 (i386-redhat-linux) using BFD version
2.11.90.0.8
/usr/local/gcc/3.2/lib/gcc-lib/i686-pc-linux-gnu/3.2/collect2 -m elf_i386
-dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o /usr/lib/crti.o /usr/local/gcc/3.2/
-L/usr/local/gcc/3.2/lib/gcc-lib/i686-pc-linux-gnu/3.2 -L/usr/local/gcc/3.2/lib/gcc-li
/tmp/ccEiIsHc.o -lfrtbegin -lg2c -lm -lgcc_s -lgcc -lc -lgcc_s -lgcc /usr/local/gcc/3.2
/usr/lib/crtn.o
```

For the purposes of this discussion, we will make a big distinction between linking to build a library and linking to build and executable. Even though these transformations have similar names, they perform very different kinds of transformations to the code.

16.2.2 Circular Dependencies and Single-Pass Linkers

Almost all linkers are single pass. This means that when linking an executable, linkers will run through the list of libraries exactly once trying to resolve symbols Ever get libraries listed in the wrong order and an executable wouldn't get built? Ever have to list the same libraries over and over again to build an executable? These are both side-effects of single pass linkers. The symbols in question are essentially jumps in the instruction code corresponding to subroutines that are defined elsewhere. When linking a final executable, all these symbols need to be resolved. When linking libraries, multiple undefined symbols are commonplace.

Having to list libraries over and over again in the link line when compiling the final executable typically indicates a circular dependency between libraries. Circular dependencies are much better kept within a single library. Even though linkers are single-pass between libraries, they exhaustively search within them.

This is important because all the files generated by Babel have a circular dependency in each Babel type. The stub makes calls on the IOR, the IOR calls the Skel, the Skel calls the Impl, but the Impl also may make calls on a Stub. Just like C++ has a `this` object, and Python has a `self`, Babel objects have a stub for them to call methods on themselves and dispatch properly through Babel's IOR layer.

16.2.3 IOR as single point of access

When building a Babelized library, its also important to note if your code has dependencies to other Babel types not in your library. These types often appear as base classes, argument types, or even exception types. Your library will need stubs corresponding to all these types, so it is best to put these in your library as well. We call these external stubs.

Many have tried to minimize replication of Babel stubs by removing the external stubs and letting the library link directly against the stubs in an external library. This is a mistake because the external library may be implemented in a different language, and the stubs may be for a different language binding. By bundling the external stubs specific to your implementation with the implementation's library, you are ensuring that the only access your library has with any other Babelized library is through the IOR. This is a good thing. The Babel IOR is the same for all language bindings and essentially forms the binary interface by which all Babel objects interact.

16.3 Dynamic vs. Static Linking

Most UNIX users are very comfortable with statically linked libraries (e.g. `libXXX.a`). Most are aware of "shared object files" in UNIX (with the form `libXXX.so`) though few actually build them. Even fewer still are familiar with dynamically linked libraries, called DLL's in Microsoft (after the common `.dll` suffix), which involve actually selecting and loading dynamic libraries at run time based on their string name. MacOSX uses the novel suffix `libXXX.dynlib`. (In most UNIX systems, including Linux and Solaris, .so "shared object files" are actually dy-

namically linked libraries.) This section serves as a quick overview of how Babel handles both static and dynamic libraries, including runtime loading.

16.3.1 Linkers and Position Independent Code (PIC)

In a static library, the linker simply copies needed compilation units from the library to the executable. The static library can subsequently be deleted with no adverse affects to the executable. This also causes common libraries to be duplicated in every executable that links against it, and for the resulting executables to be quite large.

In a shared library, the linker simply inserts in the executable enough information to find the library and load it when the executable is invoked. This typically happens before the program ever gets to `main()`. This keeps executables small and allows commonly used libraries to be reused without copying, but it also means that the executable can fail if the library is renamed, moved, deleted, or even if the user's environment changes sufficiently.

A necessary (but not sufficient) condition for shared libraries to work is that all the compilation units (`*.o`) contained must be explicitly compiled as *position independent code*(PIC). Position independent code has an added level of indirection in critical areas since details (such as addresses to jump to in subroutine calls) are not known until runtime. Even though shared libraries are very useful, PIC causes a small but measurable degradation in performance, making static linked libraries with non-PIC code a viable option for performance-critical situations.

A dynamic-linked library is a shared library with one added feature, it can be loaded explicitly by the user at runtime by passing the string name into `dlopen()`. Dynamic-linked libraries (DLL's) also require compilation as PIC, though many compilers (including GCC) have special commands for each[1].

16.3.2 Tracking Down Problems

When tracking down problems with Babel libraries, to UNIX tools **`nm`** and **`ldd`** are your friends. **`nm`** will print the list of linker symbols in a file, including details such as whether the symbol is defined or not. **`ldd`** lists dynamic dependencies of a shared libraries or executables, indicating where it will look for these symbols when loaded.

Recall the Fortran hello world example in section 16.2.1. Even though we may think this is all done with static linking, using these tools we find out the truth.

```
% ldd a.out
libg2c.so.0 => /usr/local/gcc/3.2/lib/libg2c.so.0 (0x400180000)
libm.so.6 => /lib/i686/libm.so.6 (0x4004a000)
libgcc_s.so.1 => //usr/local/gcc/3.2/lib/libgcc_s.so.1 (0x4006d000)
libc.so.6 => /lib/i686/libc.so.6 (0x40076000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Here, we clearly see that five libraries are shared libraries that will be loaded after the executable is invoked, but before we get to the main program. Some of these libraries make sense: `libg2c` is a Fortran to C support library, `libc` is the C standard library, but why is `libm` listed... its a library of transcendental functions (e.g. sin(), cos()) why would it be included? The answer becomes obvious when using ldd on `libg2c`. The fortran support library has dependencies on the math library, so our FORTRAN executable inherits that dependency too.

```
% nm a.out | grep ' U '
U __cxa_atexit@@GLIBC_2.1.3
U __libc_start_main@@GLIBC_2.0
U do_lio
U e_wsle
U exit@@GLIBC_2.0
U f_exit
U f_init
U f_setarg
U f_setsig
```

[1] `-fpic` for SO's, `-fPIC` for DLL's

```
U s_stop
U s_wsle
```

nm (and grep) shows us 11 symbols that are were left undefined in our final hello world application. A little more nm—greping about will help us find that symbols starting with `f_` are defined in libg2c.

16.4 SIDL Library Issues

As mentioned in Section 5.5, the Babel toolkit includes the SIDL runtime library. The library provides a base interface, class, and exception as the foundation. This is how Babel provides object-oriented features to non-object-oriented languages. In order to support the runtime system and build the SIDL library, it also provides DLL and Loader classes.

Babel generated code depends critically on `babel_config.h` to correctly define a lot of platform specific details. One detail that changes too frequently to encode in `babel_config.h` is whether or not the software is being compiled is position independent code (PIC). This detail is commonly added to the compilation instruction using the flags (e.g. `-fPIC -DPIC` [2]). The first flag tells the compiler to generate position independent code. The second defines the preprocessor macro `PIC`. Looking now at `babel_config.h`, we see that either `SIDL_DYNAMIC_LIBRARY` or `SIDL_STATIC_LIBRARY` are defined depending on whether or not `PIC` is defined.

As described in Section 16.3.1, Babel tends to focus on static libraries and dynamic linked libraries; not worrying much about shared libraries. The main reason is that for every last drop of performance, people would want static libraries. To support Java and Python (and the CCA model) dynamic loading is required. There's no real benefit to doing shared libraries that can't be dynamically loaded, so in developing Babel, we focus on the other two linkage situations.

16.5 Language Bindings for the `sidl` Package

The implementation and C stubs for the `sidl` package are stored in `libsidl.so` and `libsidl.a`, shared and static libraries that are installed when you install babel. You can determine the directory where these libraries are stored by running `babel-config --libdir`. Normally, running `babel-config --libdir` will yield something like `/usr/lib` or `/usr/local/lib`; however, your system administrator may have chosen a different directory by specifying a `--prefix` when they configured Babel (see Section 2.1.1). The IOR header files and C stub header files are installed in the directory shown by `babel-config --includedir`.

Babel also provides precompiled stubs for the `sidl` package for the C++, F77, F90, Java and UC++ language bindings. These libraries are also installed in `babel-config --libdir`, and they are named `libsidlstubs_cxx.so`, `libsidlstubs_ucxx.so`, Codelibsidlstubs_f77.so, and `libsidlstubs_f90.so`. Similarly named static archives and libtool `.la` files are also inalled in `babel-config --libdirst`. The header files for these languages are installed in subdirectories of `babel-config --includedir` named `Cxx`, `F77`, `F90`, and `UCxx`.

16.6 SCL Files for Dynamic Loading

If you generate a dynamic-linked library containing implementations of SIDL classes, you must also generate a SIDL Class List file (SCL file). An SCL file contains metadata about zero or more dynamic-linked libraries; for each dynamic-linked library, the SCL file has the list of SIDL classes implemented in that library. The `sidl.Loader.findLibrary` method searches SCL files when trying to find the implementation (or some other aspect) of a SIDL class.

The SCL file is an XML file with three kinds of elements. The top level element is `scl` which contains zero or more `library` elements. The `library` element has several attributes, and it contains zero or more `class` elements. The `library` element has three required attributes, `uri`, `scope` and `resolution`, and two optional attributes, `md5` and `sha1`. The `uri` is a local filename including path or a network `url` indicating where the library is stored. The `scope` attribute allows developers to suggest whether the library should be loaded in a `local` or the `global` namespace. The developer can suggest `lazy` or `now` symbol resolution using the `scope` attribute. The `md5` and `sha1` are optional message digests to confirm that the library has not been modified or replaced. The `class` element has two required elements, `name` and `desc`. The `name` field is the name of the class, and `desc` indicates what kind of

[2] The actual command to the compiler varies, `-fPIC` is understood by GCC

information is held in the library. Each class contained in the dynamic-linked library should be listed in the SCL file. For now, the only `desc` values with standardized meanings of `ior/impl`, `java` and `python/impl`. `ior/impl` indicates the dynamic-linked library contains the IOR object and implementation for the class, and `java` indicates that the library has the Java JNI wrapper object code. `python/impl` has the Python skeletons and implementation libraries.

Here is an the SCL file for the SIDL runtime library installed in `/usr/local`.

shell
```
<?xml version="1.0" ?>
<scl>
  <library uri="/usr/local/lib/libsidl.la" scope="global" resolution="now" >
    <class name="SIDL.BaseClass" desc="ior/impl" />
    <class name="SIDL.ClassInfoI" desc="ior/impl" />
    <class name="SIDL.DLL" desc="ior/impl" />
    <class name="SIDL.Loader" desc="ior/impl" />
    <class name="SIDL.Boolean" desc="java" />
    <class name="SIDL.Character" desc="java" />
    <class name="SIDL.DoubleComplex" desc="java" />
    <class name="SIDL.Double" desc="java" />
    <class name="SIDL.FloatComplex" desc="java" />
    <class name="SIDL.Float" desc="java" />
    <class name="SIDL.Integer" desc="java" />
    <class name="SIDL.Long" desc="java" />
    <class name="SIDL.Opaque" desc="java" />
    <class name="SIDL.SIDLException" desc="ior/impl" />
    <class name="SIDL.String" desc="java" />
  </library>
</scl>
```

It's worth noting that the `uri` can be a libtool metadata file (`.la`) when the library is located on the local file system or a dynamic-linked library file (`.so` or another machine dependent suffix). You cannot have a libtool `.la` when the library is remote (e.g., `ftp:` or `http:`) because libtool expects the files references in the `.la` file to be local and in particular directories.

16.7 Deployment of Babel Enabled Libraries

At this point, there is no standard — or even recommended — model for deploying Babel enabled libraries. Below are a few examples of how our developer-customers are currently packaging their code.

Server Source Only With this option your users are expected to have Babel installed on their system. In this mode, developers simply include a SIDL file and their corresponding implementation files. The user in this case must build the software, call Babel to generate the client bindings in the language of choice, and link it all together into a final application.

Client and Server Source This option tries to hide Babel as much as possible. In this mode, the developer pre-generates many different client language bindings and distributes them along with their code and the sources for the Babel runtime library. Then the user has a "batteries included" package that's ready to run out of the box. The user may not even be aware that Babel has been used unless they pay careful attention to how the package was built.

Server Libraries Only Finally, in this mode only the SIDL file and the precompiled shared library files are distributed. This is not an open-source solution, though users still need to build the language bindings to access the shared library.

Chapter 17

Creating Objects with Pre-Initialized State

or
Wrapping Native objects with Babel

Contents

17.1 Introduction to the Backdoor Initializer

Internally, every Babel object holds an implementation language specific pointer to the object's private state data. If the object is implemented in C, the pointer points to a struct named `struct package_Class__data`. In Java, the private data pointer points to a Java object of the implementation type, `Class_Impl`, which calls are made on, and which actually holds the object state. The type of the private data is entirely implementation language dependent. The data is usually created or set by the Babel objects user-implemented constructor.

With the Backdoor Initializer, Babel allows a Babel user to set this private data pointer at construction with pre-initialized state. Of course, this is very dangerous. This can only be done if the client language creating the Babel object is the same as the language the object is implemented in, and if the pre-initialized state is of the correct type. If either of these constraints is unsatisfied, the behavior of the backdoor initializer in undefined.

Object construction is a very language specific problem, and therefore this Babel feature is exposed differently in each language. However, in every language, there is some way provided for the class implementor to determine in the constructor code whether the object is being constructed normally, or if the user provided pre-initialized state. In most languages there is a second constructor provided, only in Python is an argument used to determine if the user provided pre-initialized state or not. Most class implementors will not need to do anything with the new constructor. In the case of Backdoor Initialization, usually the right thing to do in a constructor is to do nothing, since the object state was preinitialized by the client. The constructor only needs to be used if the state of the object cannot fully be represented by its private data. For example, an object that opens a TCP/IP connection during construction would almost certainly need that code in the backdoor constructor as well.

17.2 Motivation

The Backdoor Initializer is not a feature that most Babel users will need. However, there are certain cases where the Backdoor Initializer is absolutely required. The most obvious usage case is for wrapping up native objects in a Babelized interface. The allows the implementation language to access the data directly, but other languages must use the provided Babel interface. It was exactly this usage case that inspired the creation of the Backdoor Initializer.

A customer needed to use a Java visualization program to view a graph generated by a C++ library. The customer did not want to modify the Java program significantly. Instead, he created the graph data structure used by the visualizer in Java, and wrapped it in a Babel interface. He was them able to pass the Babelized object to the C++ library, which made calls on the Babel interface to add nodes and edges to the graph. When the C++ library finished, the Java visualizer was able to use the graph as if it was created natively in Java. The visualizer code did not have to be modified in any way to use the graph!

17.3 Example

In this chapter we will use a single example for all Babel supported languages. This example is taken from the wrapper regression test. In this test, there are two sidl classes, `wrapper.Data` and `wrapper.User`.

SIDL
```
package wrapper version 1.0 {

  class Data {
    void setString(in string s);
    void setInt(in int i);
  }

  class User {
     void accept(in wrapper.Data data);
  }
}
```

`wrapper.Data` wraps up some native data, which will be modified by the `wrapper.User.accept()` call. In this case, the data is just a string and an integer. In order to show the new constructor functionality, we set string called d_ctortest to "ctor was run" in the new constructor.

To reiterate, the client program creates and wraps language specific data in a `wrapper.Data` babel object. The alternate constructor code is run, which sets the d_ctortest string. The object is then passed to `wrapper.User.accept()`, which sets the data. The client program can then directly access the data and read what was set by User.accept().

17.4 The Backdoor Initializer in C

In C, the Backdoor Initializer is used through a new `_create` like static method, `_wrapObj`. `_wrapObj` takes a pointer to the private data to be wrapped (a simple struct defined in `wrapper_Data__Impl.h`).

from wrapper_Data_Impl.h:

ANSI C
```
struct wrapper_Data__data {
  /* DO-NOT-DELETE splicer.begin(wrapper.Data._data) */
  char* d_ctortest;
  char* d_string;
  int d_int;
  /* DO-NOT-DELETE splicer.end(wrapper.Data._data) */
};
```

From wrapper_Data_Impl.c; notice the new constructor `ctor2`, which is only called with backdoor initialization.

ANSI C
```
void impl_wrapper_Data__ctor2(
  /* in */ wrapper_Data self,
  /* in */ void* private_data,
  /* out */ sidl_BaseInterface *_ex) {
  /* DO-NOT-DELETE splicer.begin(wrapper.Data._ctor2) */
  struct wrapper_Data__data *dptr = (struct wrapper_Data__data *) private_data;
  dptr->d_ctorTest = "ctor was run";
  /* DO-NOT-DELETE splicer.end(wrapper.Data._ctor2) */
}

void impl_wrapper_Data_setString(
  /* in */ wrapper_Data self,
  /* in */ const char* s,
  /* out */ sidl_BaseInterface *_ex) {
  *_ex = 0;
  /* DO-NOT-DELETE splicer.begin(wrapper.Data.setString) */
  struct wrapper_Data__data *dptr =
    wrapper_Data__get_data(self);
  if (dptr) {
    dptr->d_string = "Hello World!";
  }
  /* DO-NOT-DELETE splicer.end(wrapper.Data.setString) */
}

void impl_wrapper_Data_setInt(
  /* in */ wrapper_Data self,
  /* in */ int32_t i,
  /* out */ sidl_BaseInterface *_ex) {
  /* DO-NOT-DELETE splicer.begin(wrapper.Data.setInt) */
  struct wrapper_Data__data *dptr =
    wrapper_Data__get_data(self);
  if (dptr) {
    dptr->d_int = 3;
  }
  /* DO-NOT-DELETE splicer.end(wrapper.Data.setInt) */
}
```

from the client program wraptest.c: (Note that we must include wrapper_Data_Impl.h)

ANSI C
```
#include "wrapper_User.h"
#include "wrapper_Data.h"
#include "wrapper_Data_Impl.h"
int main(int argc, char** argv) {

  sidl_BaseInterface exception = NULL;
  wrapper_Data data = NULL;
  wrapper_User user = NULL;
  struct wrapper_Data__data *d_data = NULL;
  struct wrapper_Data__data *dptr = NULL;

  /*Create the data*/
  dptr = malloc(sizeof(struct wrapper_Data__data));
  /*Wrap the data*/
  data = wrapper_Data__wrapObj(dptr, &exception);
  user = wrapper_User__create(&exception);
```

```
  ASSERT( strcmp(d_data->d_ctortest, "ctor was run") == 0);

  /* Test the data setting*/
  wrapper_User_accept(user, data, &exception);

  ASSERT( strcmp(d_data->d_string, "Hello World!") == 0);
  ASSERT( d_data->d_int == 3);

  return 0;
}
```

17.5 The Backdoor Initializer in Fortran 77

In Fortran 77, using the Backdoor Initializer is similar to using it in C. There is a special new constructor named _wrapObj that takes the private data pointer.

Of course, dynamically allocating data in FORTRAN 77 is tricky, and requires very close cooperation with the Impl class that uses the data. Most of the complexity of this example code is caused by those problems, not so much the Backdoor Initializer itself.

Since we need to store 2 strings and an integer, we create 3 sidl arrays to hold the private data. We create an opaque array of 2 elements called pdata to hold the other two arrays. Then we create a string array of 2 elements called a_string, and an integer array of 1 element called a_int. d_string is element 0 of the string array, and d_ctortest is element 1. We then place a_string into pdata as element 0, and a_int in pdata as element 1. We then call _wrapObj, which takes pdata as an in argument as the first argument, and the object we are creating, data, as an out argument as the second argument.

Notice that we don't have to include an Impl files to Fortran 77, since, there aren't actually any types.

Fairly complex, but here's the client code from wraptest.f:

Fortran 77

```
      program wraptest
      implicit none
      integer*8 data, user, pdata, backup, throwaway
      integer*8 a_string, a_int
      integer*4 d_int
      character*80 d_string
      character*80 d_ctortest
      character*80 d_silly

c     pdata is the internal data, and holds two arrays, string an int.
      call sidl_opaque__array_create1d_f(2, pdata)
      call sidl_string__array_create1d_f(2, a_string)
      call sidl_int__array_create1d_f(1, a_int)

c     initialize the data arrays
      call sidl_string__array_set1_f(a_string, 0, d_string)
      call sidl_string__array_set1_f(a_string, 1, d_ctortest)
      call sidl_int__array_set1_f(a_int, 0, d_int)

c     initilize pdata
      call sidl_opaque__array_set1_f(pdata, 0, a_string)
      call sidl_opaque__array_set1_f(pdata, 1, a_int)

      call wrapper_User__create_f(user, throwaway)
```

```
c     private data first, then the object being created
      call wrapper_Data__wrapObj_f(pdata, data, throwaway)

      call sidl_opaque__array_get1_f(pdata, 0, a_string)
      call sidl_string__array_get1_f(a_string, 1, d_ctortest)

      print *, d_ctortest

      call wrapper_User_accept_f(user, data, throwaway)

      call sidl_string__array_get1_f(a_string, 0, d_string)
      call sidl_int__array_get1_f(a_int, 0, d_int)

      print *, d_string, ' ', d_int

      call wrapper_User_deleteRef_f(user, throwaway)
      call wrapper_Data_deleteRef_f(data, throwaway)
      end
```

and the Impl side code from wrapper_Data_Impl.f

Fortran 77

```
      subroutine wrapper_Data__ctor2_fi(self, private_data, exception)
      implicit none
      integer*8 self
      integer*8 private_data
      integer*8 exception

C     DO-NOT-DELETE splicer.begin(wrapper.Data._ctor2)
      integer*8 a_string, pdata
      character*80 d_string, d_ctortest
      call sidl_opaque__array_get1_f(private_data, 0, a_string)
      call sidl_string__array_set1_f(a_string, 1, 'ctor was run')
C     DO-NOT-DELETE splicer.end(wrapper.Data._ctor2)
      end

      subroutine wrapper_Data_setString_fi(self, s, exception)
      implicit none
      integer*8 self
      character*(*) s
      integer*8 exception

C     DO-NOT-DELETE splicer.begin(wrapper.Data.setString)
      integer*8 data, a_string
      call wrapper_Data__get_data_f(self, data)
      if (data .ne. 0) then
         call sidl_opaque__array_get1_f(data, 0, a_string)
         call sidl_string__array_set1_f(a_string, 0, s)
      endif
C     DO-NOT-DELETE splicer.end(wrapper.Data.setString)
      end

      subroutine wrapper_Data_setInt_fi(self, i, exception)
      implicit none
      integer*8 self
      integer*4 i
```

```
        integer*8 exception

C       DO-NOT-DELETE splicer.begin(wrapper.Data.setInt)
        integer*8 data, a_int
        call wrapper_Data__get_data_f(self, data)
        if (data .ne. 0) then
           call sidl_opaque__array_get1_f(data, 1, a_int)
           call sidl_int__array_set1_f(a_int, 0, i)
        endif
C       DO-NOT-DELETE splicer.end(wrapper.Data.setInt)
        end
```

17.6 The Backdoor Initializer in Fortran 90

The FORTRAN 90 backdoor initializer is very similar to C. Fortran 90 also has a `_wrapObj`, but it is actually defined in the wrapper_Data_Mod.F90 file, along with the private data type definition.

Here is the private data definition from wrapper_Data_Mod.F90:

Fortran 90

```
  type wrapper_Data_priv
    sequence
    ! DO-NOT-DELETE splicer.begin(wrapper.Data.private_data)
    ! Insert-Code-Here {wrapper.Data.private_data} (private data members)
     character(len=256)     :: d_ctortest
     character(len=256)     :: d_string
     integer(kind=sidl_int) :: d_int
    ! DO-NOT-DELETE splicer.end(wrapper.Data.private_data)
  end type wrapper_Data_priv
```

Here is the client code from wraptest.F90. Notice wrapper_Data_impl is used. From wraptest.F90:

Fortran 90

```
#include "wrapper_User_fAbbrev.h"
#include "wrapper_Data_fAbbrev.h"
#include "synch_RegOut_fAbbrev.h"
#include "synch_ResultType_fAbbrev.h"

program wraptest
  use sidl
  use sidl_BaseInterface
  use wrapper_User
  use wrapper_Data
  use wrapper_Data_impl
  type(sidl_BaseInterface_t) :: throwaway_exception
  type(wrapper_Data_wrap) :: pd

  type(wrapper_Data_t) :: data
  type(wrapper_User_t) :: user

  allocate(pd%d_private_data)
  pd%d_private_data%d_int = 0
  pd%d_private_data%d_string = 'place holder'
  pd%d_private_data%d_ctortest = 'place holder'

  call new(user, throwaway_exception)
  call wrapObj(pd, data, throwaway_exception)
```

```
  print *, pd%d_private_data%d_ctortest

  call accept(user, data, throwaway_exception)

  print *, pd%d_private_data%d_string, ' ', pd%d_private_data%d_int

  call deleteRef(user, throwaway_exception)
  call deleteRef(data, throwaway_exception)
  ! Private data [should be] deallocated by the Impl dtor.

  call close(tracker, throwaway_exception)
  call deleteRef(tracker, throwaway_exception)
end program wraptest
```

Finally, the Impl code from wrapper_Data_Impl.F90:

Fortran 90

```
recursive subroutine wrapper_Data__ctor2_mi(self, private_data, exception)
  use sidl
  use sidl_BaseInterface
  use sidl_RuntimeException
  use wrapper_Data
  use wrapper_Data_impl
  implicit none
  type(wrapper_Data_t) :: self ! in
  type(wrapper_Data_wrap) :: private_data
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(wrapper.Data._ctor2)
  private_data%d_private_data%d_ctortest = 'ctor was run'
! DO-NOT-DELETE splicer.end(wrapper.Data._ctor2)
end subroutine wrapper_Data__ctor2_mi

recursive subroutine wrapper_Data_setString_mi(self, s, exception)
  use sidl
  use sidl_BaseInterface
  use sidl_RuntimeException
  use wrapper_Data
  use wrapper_Data_impl
  implicit none
  type(wrapper_Data_t) :: self ! in
  character (len=*) :: s ! in
  type(sidl_BaseInterface_t) :: exception ! out
! DO-NOT-DELETE splicer.begin(wrapper.Data.setString)
  type(wrapper_Data_wrap) :: dp
  call wrapper_Data__get_data_m(self, dp)
  dp%d_private_data%d_string = s
! DO-NOT-DELETE splicer.end(wrapper.Data.setString)
end subroutine wrapper_Data_setString_mi

recursive subroutine wrapper_Data_setInt_mi(self, i, exception)
  use sidl
  use sidl_BaseInterface
  use sidl_RuntimeException
  use wrapper_Data
```

```
  use wrapper_Data_impl
  implicit none
  type(wrapper_Data_t) :: self ! in
  integer (kind=sidl_int) :: i ! in
  type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(wrapper.Data.setInt)
  type(wrapper_Data_wrap) :: dp
  call wrapper_Data__get_data_m(self, dp)
  dp%d_private_data%d_int = i

! DO-NOT-DELETE splicer.end(wrapper.Data.setInt)
end subroutine wrapper_Data_setInt_mi
```

17.7 The Backdoor Initializer in C++

In Object Oriented languages there is no _wrapObj method exposed to the user. Instead, the same functionality is achieved simply by calling "new" on the Impl class. Interestingly, this means the constructor functionality is NOT placed in a Babel ctor method, but is, instead, actually in the default object constructor.

Here is the private data definition from wrapper_Data_Impl.hxx:

C++

```
namespace wrapper {
  class Data_impl : public virtual ::wrapper::Data

....
  public:
    char* d_string;
    int d_int;
    char* d_ctorTest;

....
  };  // end class Data_impl
} // end namespace wrapper
```

Here is the client code from wraptest.cxx. Notice wrapper_Data_Impl is included.

C++

```
#include "wrapper_User.hxx"
#include "wrapper_Data.hxx"
#include "wrapper_Data_Impl.hxx"

int main(int argc, char **argv) {
  wrapper::Data_impl data;
  wrapper::User user = wrapper::User::_create();

  ASSERT( data.d_ctorTest == "ctor was run");

  /* Test the data setting*/
  user.accept(data);

  ASSERT( data.d_string == "Hello World!");
  ASSERT( data.d_int == 3);
  return 0;
}
```

Finally, the Impl code from wrapper_Data_Impl.cxx, notice where the constructor code is placed.

C++
```
// speical constructor, used for data wrapping(required).
// Do not put code here unless you really know what you're doing!
wrapper::Data_impl::Data_impl() : StubBase(reinterpret_cast<
  void*>(::wrapper::Data::_wrapObj(this)),false) , _wrapped(true) {
  // DO-NOT-DELETE splicer.begin(wrapper.Data._ctor2)
  d_ctorTest = "ctor was run";
  // DO-NOT-DELETE splicer.end(wrapper.Data._ctor2)
}

void wrapper::Data_impl::setString_impl (
  /* in */const ::std::string& s ) {
  // DO-NOT-DELETE splicer.begin(wrapper.Data.setString)
  d_string = "Hello World!";
  // DO-NOT-DELETE splicer.end(wrapper.Data.setString)
}

void wrapper::Data_impl::setInt_impl (
  /* in */int32_t i )
{
  // DO-NOT-DELETE splicer.begin(wrapper.Data.setInt)
  d_int = 3;
  // DO-NOT-DELETE splicer.end(wrapper.Data.setInt)
}
```

17.8 The Backdoor Initializer in Java

In Object Oriented languages there is no `_wrapObj` method exposed to the user. Instead, the same functionality is achieved simply by calling "new" on the Impl class. Interestingly, this means the constructor functionality is NOT placed in a Babel `ctor` method, but is, instead, actually in the default object constructor.

Here is an excerpt from the class definition for wrapper.Data_Impl:

Java
```
  public String d_string;
  public int d_int;
  public String d_ctorTest;

  public Data_Impl(){
    d_ior = _wrap(this);
    // DO-NOT-DELETE splicer.begin(wrapper.Data._wrap)
    d_ctorTest = "ctor was run";
    // DO-NOT-DELETE splicer.end(wrapper.Data._wrap)
  }

  public void setString_Impl (
    /*in*/ java.lang.String s )
    throws sidl.RuntimeException.Wrapper
  {
    // DO-NOT-DELETE splicer.begin(wrapper.Data.setString)
    d_string = s;
    return ;
    // DO-NOT-DELETE splicer.end(wrapper.Data.setString)
  }
```

```
  public void setInt_Impl (
    /*in*/ int i )
    throws sidl.RuntimeException.Wrapper
  {
    // DO-NOT-DELETE splicer.begin(wrapper.Data.setInt)
    d_int = i;
    return ;
    // DO-NOT-DELETE splicer.end(wrapper.Data.setInt)
  }
```

Here is the client code from WrapTest.java:

Java

```
public static void main(String args[]) {
  wrapper.Data_Impl d_data = new wrapper.Data_Impl();
  wrapper.User d_user = new wrapper.User();
  System.out.println(d_data.d_ctorTest);
  d_user.accept(d_data);
  System.out.println(d_data.d_string, d_data.d_int);
}
```

17.9 The Backdoor Initializer in Python

In Object Oriented languages there is no `_wrapObj` method exposed to the user. Instead, the same functionality is achieved simply by calling "new" on the Impl class.

However, writing the Python backdoor constructor is a little trickier than Java or C++. This is because there is no overloading in Python, so multiple constructors were a problem. Instead, the class implementor needs to determine if the object is being constructed directly by the user, or through the normal Babel process. This can be achieved with an if statement. If the argument `IORself == None`, then the user has called the backdoor constructor, if `IORself != None`, it is a normal Babel construction.

Here is an excerpt from the class definition for wrapper.Data_Impl.Data:

Python

```
class Data:
  def __init__(self, IORself = None):
    if (IORself == None):
      self.__IORself = wrapper.Data.Data(impl = self)
    else:
      self.__IORself = IORself
    # DO-NOT-DELETE splicer.begin(__init__)
    if(IORself == None):
      self.d_string = "placeholder value"
      self.d_ctortest = "ctor was run"
      self.d_int = 0
    # DO-NOT-DELETE splicer.end(__init__)

  def setString(self, s):
    # DO-NOT-DELETE splicer.begin(setString)
    self.d_string = s
    # DO-NOT-DELETE splicer.end(setString)

  def setInt(self, i):
    # DO-NOT-DELETE splicer.begin(setInt)
    self.d_int = i
    # DO-NOT-DELETE splicer.end(setInt)
```

Here is the client code from WrapTest.java:

Python

```
import wrapper.User
import wrapper.Data
import wrapper.Data_Impl

if __name__ == '__main__':
  user = wrapper.User.User()
  data = wrapper.Data_Impl.Data()

  print data.d_ctortest
  user.accept(data._getStub())
  print data.d_string + " " + d_int
0
```

Chapter 18

Troubleshooting

Contents

18.1 Introduction

This appendix provides an overview of common problems that Babel users have encountered. Additional insights may be found in Chapter 19.

18.2 Common Errors

This section focuses on common errors encountered by Babel users. The errors have been separated into those related to SIDL parsing, XML parsing, and compilation.

SIDL Parsing Errors

- ***Babel: Error: when trying to resolve remaining args...Error : ÄnArgumentf̈ails to resolve as a symbol or file.*** For a symbol, Babel attempts to find it in the repository(ies) specified on the command line or, if none specified, in the default repository. Check the repository being used to ensure that XML exists for the appropriate version of the symbol. If it is not present, generate the XML for it first then try again.

XML Parsing Errors

Compilation Errors

18.3 Common Warnings

This section focuses on common warnings encountered by Babel users. Again, warnings have been separated into those related to SIDL parsing, XML parsing, and compilation.

SIDL Parsing Warnings

- ***Babel: Warning: When creating repository...File R̈epository+Fileïs not a repository directory".*** First verify that the specified directory is actually a repository directory. That is, that it contains symbol interfaces defined by XML files. If not, correct the repository option then try again.

XML Parsing Warnings

Compilation Warnings

Chapter 19

Lessons Learned

Contents

19.1 Introduction

This appendix focuses on providing tips, tricks, and advice submitted by Babel/SIDL users. We have generally provided the information verbatim.

19.2 Compilation Consistency is Key

Steve Smith, 24 September 2001

Basically "be consistent" is the biggest lesson we found.

When compiling C++ codes, you may have conflicts if you use different compile options. Under KCC we found -no_exceptions caused problems if parts were compiled with/without the flag. There are most likely other compile flags which turn features on/off which would cause similar problems. This caused a core dump immediately when core file was loaded. This is somewhat obvious but if you are linking together several different codes from a variety of developers you need to examine the compile flags very carefully. This problem is probably more likely with C++ due to the greater number of code generation options (e.g. RTTI, exceptions etc).

A much more subtle problem occurred when we had a C shared library which called functions in a C++ shared library. We initially used gcc to create the C shared library and KCC to create the C++ shared library. The application would core dump when a dynamic cast was attempted. This was solved by using the `cc` compiler wrapper that is part of the KCC distribution (which uses the native `cc`). So you need to be aware of not only what is in your .so and how it is compiled but all the .so's that you are using.

If you have several versions of a library, say during a debugging process, make sure you are using the correct versions of things. The `ldd` command is very useful for making sure you getting the shared libraries that you think you should be linking to. Along these lines, keep your LD_LIBRARY_PATH as simple as possible when debugging.

In retrospect this does not look like a large number of problems, but figuring out the second problem took a long time since I focused on how the C++ library was being created rather than where the real problem was being introduced. It wasn't until after I had exhausted a long list of other potential conflicts that I started messing with the C library compilation.

Part IV

Appendices

Appendix A

Reserved Words

Contents

A.1 Introduction

This appendix lists SIDL's reserved words. Other words and constructs that are problematic in particular language bindings are also listed.

A.2 Reserved Words

Table A.1 lists all the words that are part of the SIDL grammar and cannot be used as a package, enum, interface, class, or argument name.

A.3 Suggested Things To Avoid

Since SIDL maps onto many other languages there are a great number of words and constructs that are harmless in SIDL, but cause great trouble in generated language bindings. We list known problems in Table A.2.

In addition, the following should be avoided:

- Reserved words in all of the supported languages. This is a long list only some of which appear here.
- Methods with the same name as a class (this is a constructor in C++).
- Packages, Classes, Interfaces, Methods or Arguments that differ only by case. Not all languages are case sensitive but, since Babel's focus is language interoperability, Babel must make allowances.

Table A.1: SIDL Reserved Words

RESERVED WORD	ROLE
abstract	optional modifier for *class*
array	datatype
bool	builtin datatype
char	builtin datatype
class	user defined datatype
copy	argument modifer
dcomplex	builtin datatype
double	builtin datatype
enum	user defined datatype
extends	inheritance mode
fcomplex	builtin datatype
final	package and method modifier
float	builtin datatype
implements	inheritance mode
implements-all	inheritance mode
import	bring other packages into current scope
in	argument mode
inout	argument mode
int	builtin datatype
interface	user defined datatype
local	method modifier
long	builtin datatype
nonblocking	method modifier
oneway	method modifier
opaque	builtin datatype
out	argument mode
package	scoping construct
static	method modifier
string	builtin datatype
throws	exception declaration
version	assign version number to package
void	declares method as not returning a type

Table A.2: Other words/constructs to avoid

WORD	C	C++	Java	Python	word	C	C++	Java	Python
abstract			X		lambda				X
and		X		X	long	X	X	X	
and_eq		X			mutable		X		
asm	X	X			namespace		X		
assert				X	native			X	
auto	X	X			new		X	X	
bitand		X			not		X		X
bitor		X			not_eq		X		
bool		X			null			X	
boolean			X		operator		X		
break	X	X	X	X	or		X		X
case	X	X	X		or_eq		X		
catch		X	X		package			X	
char	X	X	X		pass				X
class		X	X		print				X
compl		X			private		X	X	
const	X	X	X		protected		X	X	
const_cast		X			public		X	X	
continue	X	X	X	X	raise				X
def				X	register	X	X		
default	X	X	X		reinterpret_cast		X		
del				X	return	X	X	X	X
delete		X			short	X	X	X	
do	X	X	X		signed	X	X		
double	X	X	X		sizeof	X	X		
dynamic_cast		X			static	X	X	X	
elif				X	static_cast		X		
else	X	X	X	X	strictfp			X	
enum	X	X			struct	X	X		
except				X	super			X	
exec				X	switch	X	X	X	
explicit		X			synchronized			X	
export		X			template		X		
extends			X		this		X	X	
extern	X	X			throw		X	X	
false		X	X		throws			X	
final			X		transient			X	
finally			X	X	true		X	X	
float	X	X	X		try		X	X	X
for	X	X	X	X	typedef	X	X		
friend		X			typeid		X		
from				X	typename		X		
global				X	union	X	X		
goto	X	X	X		unsigned	X	X		
if	X	X	X	X	using		X		
implements			X		virtual		X		
import			X		void	X	X	X	
inline		X			volatile	X	X	X	
instanceof			X		wchar_t		X		
int	X	X	X		while	X	X	X	X
interface			X		xor		X		
is				X	xor_eq		X		

Appendix B

SIDL Grammar

Contents

B.1 Introduction

This appendix provides an overview of the Scientific Interface Definition Language (SIDL) grammar. For simplicity, the grammar is described in extended BNF.

B.2 Backus-Naur Form

The grammar described here was extracted from the JavaCC productions defined in the Babel source code. Since the comments associated with the productions appeared to be sufficiently descriptive, they have been retained to serve as the explanation of the key productions.

```
/*
 * The following lexical tokens are ignored.
 */
SKIP : {
   < " " >
 | < "\n" >
 | < "\r" >
 | < "\t" >
 | < "//" (~["\n","\r"])* ("\n" | "\r" | "\r\n") >
 | < "/**/" >
 | < "/*" (~["*"])+ "*" ("*" | ~["*","/"] (~["*"])* "*")* "/" >
   { checkComment(image, input_stream.getBeginLine(),
                  input_stream.getEndLine()); }
 | < "[" >
 | < "]" >
}

/*
 * The following lexical states define the transitions necessary to
 * parse documentation comments.  Documentation comments may appear
 * anywhere in the file, although they are only saved if they preceed
```

```
 * definition or method productions.  Documentation comments are
 * represented by "special tokens" in the token list.
 */
SPECIAL_TOKEN : {
   < T_COMMENT : "/**" > : BEGIN_DOC_COMMENT
}

<BEGIN_DOC_COMMENT> SKIP : {
   < " " >
 | < "\t" >
 | < "*/" >                    : DEFAULT
 | < ("\n" | "\r" | "\r\n") > : LINE_DOC_COMMENT
 | < "" >                      : IN_DOC_COMMENT
}

<LINE_DOC_COMMENT> SKIP : {
   < " " >
 | < "\t" >
 | < "*/" >       : DEFAULT
 | < "*" (" ")?> : IN_DOC_COMMENT
 | < "" >         : IN_DOC_COMMENT
}

<IN_DOC_COMMENT> SPECIAL_TOKEN : {
   < "*/" >                    { trimMatch(matchedToken); } : DEFAULT
 | < ("\n" | "\r" | "\r\n") > { trimMatch(matchedToken); } : LINE_DOC_COMMENT
}

<IN_DOC_COMMENT> MORE : {
   < ~[] >
}

/*
 * The following keywords are the lexical tokens in the SIDL grammar.
 */
TOKEN : {
   < T_ABSTRACT       : "abstract" >
 | < T_CLASS          : "class" >
 | < T_COPY           : "copy" >
 | < T_ENUM           : "enum" >
 | < T_EXTENDS        : "extends" >
 | < T_IMPORT        : "import" >
 | < T_IN             : "in" >
 | < T_INOUT          : "inout" >
 | < T_FINAL          : "final" >
 | < T_IMPLEMENTS     : "implements" >
 | < T_IMPLEMENTS_ALL : "implements-all" >
 | < T_INTERFACE      : "interface" >
 | < T_LOCAL          : "local" >
 | < T_ONEWAY         : "oneway" >
 | < T_OUT            : "out" >
 | < T_PACKAGE        : "package" >
 | < T_REQUIRE        : "require" >
 | < T_STATIC         : "static" >
```

```
 | < T_THROWS         : "throws" >
 | < T_VERSION        : "version" >
 | < T_VOID           : "void" >

 | < T_ARRAY          : "array" >
 | < T_RARRAY         : "rarray" >
 | < T_BOOLEAN        : "bool" >
 | < T_CHAR           : "char" >
 | < T_DCOMPLEX       : "dcomplex" >
 | < T_DOUBLE         : "double" >
 | < T_FCOMPLEX       : "fcomplex" >
 | < T_FLOAT          : "float" >
 | < T_INT            : "int" >
 | < T_LONG           : "long" >
 | < T_OPAQUE         : "opaque" >
 | < T_STRING         : "string" >

 | < T_IDENTIFIER     : <T_LETTER> (<T_LETTER> | <T_DIGIT> | "_")* >
 | < T_VERSION_STRING : <T_INTEGER> ("." <T_INTEGER>)+ >
 | < T_INTEGER        : (["-","+"])? (<T_DIGIT>)+ >
 | < T_DIGIT          : ["0"-"9"] >
 | < T_LETTER         : ["a"-"z","A"-"Z"] >

 | < T_CLOSE_ANGLE    : ">" >
 | < T_CLOSE_CURLY    : "}" >
 | < T_CLOSE_PAREN    : ")" >
 | < T_COMMA          : "," >
 | < T_EQUALS         : "=" >
 | < T_OPEN_ANGLE     : "<" >
 | < T_OPEN_CURLY     : "{" >
 | < T_OPEN_PAREN     : "(" >
 | < T_SEMICOLON      : ";" >
 | < T_SCOPE          : "." >

 | < T_COLUMN_MAJOR   : "column-major" >
 | < T_ROW_MAJOR      : "row-major" >

 | < T_CATCH_ALL      : ~[] >
}

/**
 * A SIDL Specification contains zero or more version productions followed
 * by zero or more import productions followed by zero or more package
 * productions followed by the end-of-file.  Before leaving the specification
 * scope, resolve all references in the symbol table.
 */
Specification ::= ( Require )* ( Import )* ( Package )* <EOF>

/**
 * A SIDL Require production begins with a "require" token and is followed
 * by a scoped identifer, a "version" token, and a version number.  The
 * scoped identifier must be not defined.  The version number is specified
 * in the general form "V1.V2...Vn" where Vi is a non-negative integer.
 */
```

```
Require ::=
  <T_REQUIRE> ScopedIdentifier
  <T_VERSION> ( <T_INTEGER> | <T_VERSION_STRING> ) <T_SEMICOLON>

/**
 * A SIDL Import production begins with an "import" token and is followed
 * by a scoped identifier which is optionally followed by a "version" token
 * and a version number.  The scoped identifier must be defined and it must
 * be a package.  The version number is specified in the general form
 * "V1.V2...Vn" where Vi is a non-negative integer.  A particular package
 * may only be included in one import statement.  The import package name
 * is added to the default search path.  At the end of the parse, any import
 * statements that were not used to resolve a symbol name are output as
 * warnings.
 */
Import ::=
  <T_IMPORT> ScopedIdentifier
  [ <T_VERSION> ( <T_INTEGER> | <T_VERSION_STRING>) ] <T_SEMICOLON>

/**
 * The SIDL package specification begins with a "package" token followed by
 * a scoped identifier.  The new package namespace begins with an open curly
 * brace, a set of zero or more definitions, and a close curly brace.  The
 * closing curly brace may be followed by an optional semicolon.  The package
 * identifier must have a version defined for it, and it must not have been
 * previously defined as a symbol or used as a forward reference.  The parent
 * of the package must itself be a package and must have been defined.  The
 * symbols within the curly braces will be defined within the package scope.
 */
Package ::=
  [ <T_FINAL> ] <T_PACKAGE> ScopedIdentifier
  [ <T_VERSION> ( <T_INTEGER> | <T_VERSION_STRING> ) ]
  <T_OPEN_CURLY> ( Definition )* <T_CLOSE_CURLY> [ <T_SEMICOLON> ]

/**
 * A SIDL Definition production consists of a class, interface, enumerated
 * type, or package.
 */
Definition ::= ( Class | Enum | Interface | Package )

/**
 * A SIDL class specification begins with an optional abstract keyword
 * followed by the class token followed by an identifier.  The abstract
 * keyword is required if and only if there are abstract methods in the
 * class.  The class keyword is followed by an identifer.  The identifier
 * string may not have been previously defined, although it may have been
 * used as a forward reference.  The identifier string may be preceeded
 * by a documentation comment.  A class may optionally extend another class;
 * if no class is specified, then the class will automatically extend the
 * SIDL base class (unless it is itself the SIDL base class).  Then parse
 * the implements-all and implements clauses.  The interfaces parsed during
 * implements-all are saved in a set and then all those methods are defined
 * at the end of the class definition.  The methods block begins with an
 * open curly-brace followed by zero or more methods followed by a close
```

```
 * curly-brace and optional semicolon.
 */
Class ::=
  [ <T_ABSTRACT> ] <T_CLASS> Identifier
  [ <T_EXTENDS> ScopedIdentifier ]
  [ <T_IMPLEMENTS_ALL> AddInterface ( <T_COMMA> AddInterface )* ]
  [ <T_IMPLEMENTS> AddInterface ( <T_COMMA> AddInterface )* ]
  <T_OPEN_CURLY> ( ClassMethod )* <T_CLOSE_CURLY> [ <T_SEMICOLON> ]

/**
 * The SIDL enumeration specification begins with an "enum" token followed by
 * an identifier.  The enumerator list begins with an open curly brace, a set
 * of one or more definitions, and a close curly brace.  The closing curly
 * brace may be followed by an optional semicolon.  The enumeration symbol
 * identifier must have a version defined for it, and it must not have been
 * previously defined as a symbol.  Forward references are not allowed for
 * enumerated types.  This routine creates the enumerated class and then
 * grabs the list of enumeration symbols and their optional values.
 */
Enum ::=
  <T_ENUM> Identifier <T_OPEN_CURLY> Enumerator ( <T_COMMA> Enumerator )*
  <T_CLOSE_CURLY> [ <T_SEMICOLON> ]

/**
 * The SIDL enumerator specification consists of an identifier followed
 * by an optional assignment statement beginning with an equals and followed
 * by an integer value.  This routine adds the new enumeration symbol to
 * the list and then returns.
 */
Enumerator ::= Identifier [ <T_EQUALS> <T_INTEGER> ]

/**
 * A SIDL interface specification begins with the interface token followed
 * by an identifier.  An interface may have an extends block consisting of
 * a comma-separated sequence of interfaces.  The methods block begins with
 * an open curly-brace followed by zero or more methods followed by a close
 * curly-brace and optional semicolon.  Interfaces may be preceeded by a
 * documentation comment.  The identifier string may not have been previously
 * defined, although it may have been used as a forward reference.  If the
 * interface does not extend another interface, then it must extend the base
 * SIDL interface (unless, of course, this is the definition for the base
 * SIDL interface).
 */
Interface ::=
  <T_INTERFACE> Identifier [ <T_EXTENDS> AddInterface
  ( <T_COMMA> AddInterface )* ]
  <T_OPEN_CURLY> ( InterfaceMethod )* <T_CLOSE_CURLY> [ <T_SEMICOLON> ]

/**
 * This production parses the next scoped identifier and validates that
 * the name exists and is an interface symbol.  Then each of its methods
 * are checked for validity with the existing methods.  If everything
 * checks out, then the new interface is added to the existing object.
 */
```

```
AddInterface ::= ScopedIdentifier

/**
 * This production parses the SIDL method description for a class method.
 * A class method may start with abstract, final, or static.  An error is
 * thrown if the method has already been defined in the class object or if
 * the method name is the same as the class name.  An error is also thrown
 * if a method has been defined in a parent class and (1) the signatures
 * do not match, (2) either of the methods is static, (3) the existing method
 * is final, or (4) the new method is abstract but the existing method was
 * not abstract.
 */
ClassMethod ::= [ ( <T_ABSTRACT> | <T_FINAL> | <T_STATIC> ) ] Method

/**
 * This method parses a SIDL method and then checks whether it can be
 * added to the interface object.  An error is thrown if the method has
 * already been added to the interface object or if the method name is
 * the same as the interface name.  An error is also thrown if a previous
 * method was defined with the same name but a different signature.
 */
InterfaceMethod ::= Method

/**
 * The SIDL method production has a return type, a method identifier,
 * an optional argument list, an optional communication modifier, and
 * an optional throws clause.  The return type may be void (no return
 * type) or any valid SIDL type.  The method is built piece by piece.
 */
Method ::=
  ( <T_VOID> | [ <T_COPY> ] Type() ) Identifier [ <T_IDENTIFIER> ]
  <T_OPEN_PAREN> [ Argument ( <T_COMMA> Argument )* ] <T_CLOSE_PAREN>
  [ <T_LOCAL> | <T_ONEWAY> ] [ <T_THROWS> ScopedIdentifier
  ( <T_COMMA> ScopedIdentifier )* ] <T_SEMICOLON>

/**
 * Parse a SIDL argument.  Arguments begin with an optional copy modifier
 * followed by in, out, or inout followed by a type and a formal argument.
 * The argument is returned on the top of the argument stack.  This routine
 * also checks that the copy modifier is used only for symbol objects.  For
 * all other types, copy is redundant.
 */
Argument ::= [ <T_COPY> ] ( <T_IN> | <T_OUT> | <T_INOUT> )
         (Type Identifier | Rarray)

/**
 * A SIDL type consists of one of the standard built-in types (boolean,
 * char, dcomplex, double, fcomplex, float, int, long, opaque, and string),
 * a user-defined type (interface, class, or enum), or an array.  This
 * production parses the type and pushes the resulting type object on
 * the top of the argument stack.
 */
Type ::=
  ( <T_BOOLEAN>
```

```
  | <T_CHAR>
  | <T_DCOMPLEX>
  | <T_DOUBLE>
  | <T_FCOMPLEX>
  | <T_FLOAT>
  | <T_INT>
  | <T_LONG>
  | <T_OPAQUE>
  | <T_STRING>
  | Array
  | SymbolType )

/**
 * Parse an array construct and push the resulting type and ordering
 * on top of the stack.  Only dimensions one through MAX_ARRAY_DIM
 * (inclusive) are supported.
 */
Array ::=
  <T_ARRAY> <T_OPEN_ANGLE> Type [ <T_COMMA> ( <T_INTEGER>
  [ <T_COMMA> ( <T_COLUMN_MAJOR> | <T_ROW_MAJOR> ) ]
  | ( <T_COLUMN_MAJOR>| <T_ROW_MAJOR> ) ) ] <T_CLOSE_ANGLE>

/**
 * Parse an rarray construct and push the resulting type and ordering
 * on top of the stack.  Only dimensions one through MAX_ARRAY_DIM
 * (inclusive) are supported.  And don't forget the indicies!
 */
Rarray ::= <T_RARRAY> <T_OPEN_ANGLE> Type [ <T_COMMA> <T_INTEGER> ]
       <T_CLOSE_ANGLE> Identifier
       <T_OPEN_PAREN> Identifier ( <T_COMMA Identifier )*
       <T_CLOSE_PAREN>

/**
 * This production parses a scoped identifier and verifies that it is
 * either a forward reference or a symbol that may be used as a type
 * (either an enum, an interface, or a class).
 */
SymbolType ::= ScopedIdentifier

/**
 * All SIDL scoped names are of the general form "ID ( . ID )*".  Each
 * identifier ID is a string of letters, numbers, and underscores that
 * must begin with a letter.  The scope resolution operator "." separates
 * the identifiers in a name.
 */
ScopedIdentifier ::= Identifier ( <T_SCOPE> Identifier )*

/**
 * A SIDL identifier must start with a letter and may be followed by any
 * number of letters, numbers, or underscores.  It may not be a reserved
 * word in any of the SIDL implementation languages (e.g., C or C++).
 */
Identifier ::= <T_IDENTIFIER>
```

Appendix C

Extensible Markup Language (XML)

Contents

C.1 Introduction

This appendix describes the XML representation of SIDL interfaces. Since the format of an XML file is dictated by a Document Type Declaration (DTD) file, the description will focus on the DTD associated with SIDL.

C.2 SIDL Document Type Declaration (DTD)

Babel relies on several DTDs to describe and enforce the layout of conformant XML files. The DTD of primary importance for Babel is `SIDL.dtd` because it describes the requisite tags and attributes corresponding to SIDL files. The contents of the DTD are given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
   File:        sidl.dtd
   Package:     SIDL XML
   Revision:    @(#) $Id: SIDL.dtd 1257 2005-12-12 23:26:51Z epperly $
   Description: DTD for the SIDL XML database representation

   Copyright (c) 2000-2005, The Regents of the University of Calfornia.
   Produced at the Lawrence Livermore National Laboratory.
   Written by the Components Team <components@llnl.gov>
   UCRL-CODE-2002-054
   All rights reserved.

   This file is part of Babel. For more information, see
   http://www.llnl.gov/CASC/components/. Please read the COPYRIGHT file
   for Our Notice and the LICENSE file for the GNU Lesser General Public
   License.

   This program is free software; you can redistribute it and/or modify it
   under the terms of the GNU Lesser General Public License (as published by
```

```
   the Free Software Foundation) version 2.1 dated February 1999.

   This program is distributed in the hope that it will be useful, but
   WITHOUT ANY WARRANTY; without even the IMPLIED WARRANTY OF
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the terms and
   conditions of the GNU Lesser General Public License for more details.

   You should have recieved a copy of the GNU Lesser General Public License
   along with this program; if not, write to the Free Software Foundation,
   Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

   This file describes the DTD for a SIDL symbol represented in XML format.
   The root element is <Symbol>.

   PUBLIC ID "-//CCA//sidl Symbol DTD v1.2//EN"
-->

<!--
   Symbol Element

   Symbol is the root element for all SIDL XML schema.  The Symbol contains a
   SymbolName (fully qualified symbol name and version), Metadata, Comment,
   and one of Class, Enumeration, Interface, or Package.
-->

<!ENTITY % symbols "Class | Enumeration | Interface | Package">
<!ELEMENT Symbol (SymbolName, Metadata, Comment, (%symbols;))>

<!--
   SymbolName Element

   A SymbolName represents a fully qualified symbol name along with its
   version.  It is of the form:

   <SymbolName name="sidl.SomeName" version="1.3.4"/>
-->

<!ELEMENT SymbolName EMPTY>
<!ATTLIST SymbolName name     CDATA #REQUIRED
                     version  CDATA #REQUIRED>

<!--
   Metadata Element

   The Metadata element contains any useful descriptive data about the symbol.
   The time and date of creation is required, but all other information is
   optional.  The date and time must follow the ISO-8601 standard.  The
   entries in the metadata element are (key,value) pairs.
-->

<!ELEMENT Metadata (MetadataEntry)*>
<!ATTLIST Metadata date CDATA #REQUIRED>

<!ELEMENT MetadataEntry EMPTY>
```

```
<!ATTLIST MetadataEntry key   CDATA #REQUIRED
                        value CDATA #REQUIRED>

<!--
   Comment Element

   Comment elements support a very simple HTML description using the
   html-lite.dtd HTML subset.  See html-lite.dtd for more details.
-->

<!ENTITY % html-lite PUBLIC "-//CCA//sidl HTML DTD v1.0//EN" "html-lite.dtd">
%html-lite;

<!ELEMENT Comment %html-block;>

<!--
   Package Element

   The Package element contains the symbols that exist within a package.
   In the PackageSymbol element, note that the name is relative to the
   package (thus, sidl.Class is represented by Class within package sidl).

   A true final attribute indicates that this package is not reentrant. It
   defaults to true to handle old XML files. In previous versions, all
   packages were non-reentrant.
-->

<!ELEMENT Package (PackageSymbol)*>
<!ATTLIST Package final (false | true) "true">

<!--
If the version attribute isn't provided, the symbol has the same version
as the containing package. This is to provide backward compatibility with
previous released versions of the DTD. Someday the version may become
REQUIRED, so always include it.
-->
<!ELEMENT PackageSymbol EMPTY>
<!ATTLIST PackageSymbol name CDATA                                  #REQUIRED
                        type (class | enum| interface | package) #REQUIRED
                        version CDATA                               #IMPLIED>

<!--
   Enumeration Element

   The Enumeration element consists of a collection of Enumerator elements
   that describe a relative symbol name, its integer value, and whether the
   value was assigned by the parser or in the SIDL input file.
-->

<!ELEMENT Enumeration (Enumerator)+>

<!ELEMENT Enumerator (Comment)?>
<!ATTLIST Enumerator name     CDATA          #REQUIRED
                     value    CDATA          #REQUIRED
```

```
                    fromuser (false | true) #REQUIRED>

<!--
   Class Element

   The Class element consists of a class extended by this class, a block
   of interfaces implemented by this class, and a block of methods declared
   or defined by this class.  The methods block does not include methods
   declared or defined by parents.  The elements AllParentInterfaces and
   AllParentClasses include all parents of this class.
-->
<!ELEMENT Class (Extends, ImplementsBlock,
                 AllParentClasses, AllParentInterfaces,
                 MethodsBlock, AssertionList?)>
<!ATTLIST Class abstract (false | true) #REQUIRED>

<!ELEMENT Extends (SymbolName)?>

<!ELEMENT ImplementsBlock (SymbolName)*>

<!--
   Interface Element

   The Interface element constsis of a block of interfaces that this
   interface extends (element ExtendsBlock) and a block of methods
   declared by this interface (element MethodsBlock).  The methods block
   element contains only those methods declared or re-declared by this
   interface and does not include all those methods defined by the
   parent interfaces.  The AllParentInterfaces element block includes
   all parent interfaces that this interface extends.
-->

<!ELEMENT Interface (ExtendsBlock,  AllParentInterfaces,
                     MethodsBlock, AssertionList?)>

<!ELEMENT ExtendsBlock (SymbolName)*>

<!--
   AllParentClasses and AllParentInterfaces Elements

   These elements define a collection of zero or more SymbolName elements
   that are the parent classes and parent interfaces of a class or interface.
-->

<!ELEMENT AllParentClasses (SymbolName)*>

<!ELEMENT AllParentInterfaces (SymbolName)*>


<!--
   MethodsBlock Element

   The MethodsBlock element defines a collection of zero or more methods
   that belong to a SIDL interface or class.
```

```
-->

<!ELEMENT MethodsBlock (Method)*>

<!--
   Method Element

   The Method element defines the signature for a single method in a class or
   interface.  The name of the method is obtained from the shortname.  If
   method name overloading is not supported, the extension is appended to the
   short name to build the method name.
-->

<!ELEMENT Method (Comment, Type, ArgumentList, ThrowsList, ImplicitThrowsList, AssertionLis
<!ATTLIST Method shortname     CDATA                                  #REQUIRED
                 extension     CDATA                                  #REQUIRED
                 copy          (false  | true)                        #REQUIRED
                 definition    (normal | abstract | final | static) #REQUIRED
                 communication (normal | local    | oneway)         #REQUIRED>

<!ELEMENT ArgumentList (Argument)*>

<!ELEMENT ThrowsList (SymbolName)*>

<!ELEMENT ImplicitThrowsList (SymbolName)*>

<!ELEMENT AssertionList (Assertion)*>

<!--
   Argument Element

   The SIDL Argument element defines a SIDL method argument.
-->

<!ELEMENT Argument (Type)>
<!ATTLIST Argument copy (false | true)     #REQUIRED
                   mode (in | inout | out) #REQUIRED
                   name CDATA              #REQUIRED>

<!--
   Type Element

   The Type element describes a SIDL type, which may be a built-in type
   such as boolean or int, an array, or a user-defined symbol.  If the
   type description is a primitive type, then no sub-elements are allowed.
   If the type is a symbol, then the single sub-element must be a symbol
   name.  If the type is an array, then the single sub-element must be
   an array element
-->

<!ELEMENT Type (SymbolName | Array)?>
<!ATTLIST Type type (void     | boolean | char    | dcomplex | double |
                    fcomplex | float   | integer | long     |
                    opaque   | string  | symbol  | array    ) #REQUIRED>
```

```
<!ELEMENT Index EMPTY>
<!ATTLIST Index name CDATA #REQUIRED>

<!ELEMENT Array (Type?,Index?)>
<!ATTLIST Array order (unspecified | column-major | row-major) #REQUIRED
                dim   CDATA "0" >


<!--
   Assertion Element

   The SIDL Assertion element defines a SIDL assertion.
-->
<!ELEMENT Assertion (Comment, AssertionExpression)>
<!ATTLIST Assertion tag  CDATA                                  #REQUIRED
                    type ( invariant | require     | require_else
                         | ensure    | ensure_then )           #REQUIRED>

<!--
   AssertionExpression Element

   The SIDL Assertion Expression element defines a valid assertion expression.
-->
<!ELEMENT AssertionExpression ( BinaryExpression | ComplexNumber | MethodCall
                              | SimpleExpression | UnaryExpression)>
<!ATTLIST AssertionExpression parens (true | false)              "false">

<!--
   BinaryExpression Element

   The SIDL Binary Expression element defines a binary assertion expression.
-->
<!ELEMENT BinaryExpression (AssertionExpression, AssertionExpression)>
<!ATTLIST BinaryExpression op ( and          | divide        | equals
                              | expon        | greater_than  | greater_equal
                              | iff          | implies       | less_equal
                              | less_greater | less_than     | minus
                              | modulus      | multiply      | not_equal
                              | or           | plus          | power
                              | remainder    | shift_left    | shift_right
                              | xor
                              )                               #REQUIRED>

<!--
   ComplexNumber Element

   The  Complex Number element defines a complex number assertion expression.
-->
<!ELEMENT ComplexNumber EMPTY>
<!ATTLIST ComplexNumber type      (float | double)         #REQUIRED
                        real      CDATA                    #REQUIRED
                        imaginary CDATA                    #REQUIRED>
```

```
<!--
   MethodCall Element

   The SIDL Method Call element defines a method call assertion expression.
   Note that any arguments must be within the scope of the assertion.  For
   invariants, expressions can only contain literals (i.e., NO state or
   attributes).  For methods, expressions can also contain any arguments
   that are being passed to the method.
-->
<!ELEMENT MethodCall (AssertionExpression*)>
<!ATTLIST MethodCall name CDATA                              #REQUIRED>

<!--
   SimpleExpression Element

   The  Simple Expression element defines expressions that do not have
   operators; namely, identifiers and literals.  Note the only valid literals
   are boolean, character, double, float, integer, and string and identifiers
   are symbols.
-->
<!ELEMENT SimpleExpression (Type)>
<!ATTLIST SimpleExpression etype (constant | identifier)  #REQUIRED
                           value CDATA                    #REQUIRED>

<!--
   UnaryExpression Element

   The SIDL Unary Expression element defines a unary assertion expression.
-->
<!ELEMENT UnaryExpression (AssertionExpression)>
<!ATTLIST UnaryExpression op  (complement | is | minus | not | plus )
                                                          #REQUIRED>
```

Babel assumes that comments will conform to the HTML-lite comment format. So, Babel relies on `comment.dtd` to validate whether SIDL documentation comments follow the HTML-lite comment format, which is described in `html-lite.dtd`. The most current versions of all of these DTDs can also be found in the source distribution in the `babel/compiler/gov/llnl/babel/dtds` directory.

NOTE: Any XML interface description that complies with the SIDL DTD can be used as input to Babel.

Appendix D

Glossary

abstract

OOP concept: Abstract describes something that is declared but not fully defined. For example, an abstract method is a method that is declared as a part of a class, but has no implementation. It cannot be called, it is only meant to be inherited by derived classes.

SIDL keyword: Abstract is an optional modifier for both `class`es and `method`s. An abstract method is a method that has no implementation, it's a way of declaring a method that every subclass must implement for itself. An abstract class has one or more abstract methods, and therefore cannot be instantiated.

array

Datastructure: An array is a fixed size, numerically indexed, set of variables. Arrays have in language support in almost all modern programming languages.
Babel: Babel has built in support for arrays of every data type, including objects. Babel allows these arrays such that they may be shared by differing languages.

BLAS

Basic Linear Algebra Subprograms. BLAS is a famous library for doing matrix and vector algebra. More information may be found at: http://www.netlib.org/blas/

Babel Object Server

A Babel Object Server (BOS) is a network server process or thread that provides babel objects via Remote Method Invocation (RMI). Normally a BOS is run as a background thread on a normal Babel process to allow the process to publish objects for access by RMI enabled clients. There is not a single protocol that a BOS must use to communicate over Babel RMI, but clients and BOSs must use the same protocol if they are expected to communicate.

BNF

BackusNaur Form. BNF is a formal way to describe computer languages and other formal languages.

bool

Definition: bool is a short form of the word boolean. A boolean is a logical data type that holds 1 bit of data, i.e. it is either true or false. It is used for Boolean Algebra.
SIDL keyword: bool is a data type built into SIDL, an instance of which is either true or false. For efficiency sake, the underlying storage of bool is not 1 bit.

borrowed arrays

Babel: A borrowed array is a SIDL array that does not manage its own data. The data is provided by some third party, who is also in charge of deallocating the data. It is useful for sending data through Babel, but the developer must beware in case the third party deallocates the array data before the program has finished with it.

CCA

Common Component Architecture http://www.ccaforum.org/

char

Definition: char is a short form of the word character. A character is a letter, number, puctuation mark, or other such symbol use in writing. In programming, a character is often defined by the 8 bit ASCII encoding.
SIDL keyword: char is a data type built into SIDL. It stores 1 byte of data, or enough for 1 ASCII character.

class

OOP concept: A class is a definition for a particular kind of object. It may define the data and methods that will be included in an actual instance of the object.
SIDL keyword: class is a SIDL keyword. In SIDL a class definition only defines methods. Methods may be static or instance methods. (They are instance methods by default.) If any instance method in a class is declared abstract, the class cannot be instantiated as an object, and is called an abstract class. Otherwise, it can be instantiated and is called a concrete class.

concrete class

OOP concept: A concrete class is a class where all the class's instance methods have implementations. (ie. there are no abstract methods) A concrete class may be instantiated as an object.

COM

Common Object Model http://www.microsoft.com/ Microsoft's IDL based language interoperability suite.

component

OOP concept: Components are "plug-and-play" software libraries designed with standard, clearly defined interfaces. They are the epitome of modular design. Because components communicate only through well-defined interfaces, when an application needs to be modified, a single component can be modified (or exchanged for a similar component), without fear of disrupting the other components making up the application.

component architecture

OOP concept: A component architecture defines the specifics of setting up a system for programming with components in that architecture. For example, how components are imported and how they communicate are some of the questions that must be answered in a component architecture design.

copy

SIDL keyword: copy is a SIDL keyword. It is planned that in future version of babel it will be used as a parameter modifier for parameters passed to RMI functions, currently however, this feature is unimplemented.

CORBA

Common Object Request Broker Architecture http://www.omg.org CORBA allows different programs by different vendors to communicate though an IDL interface specification. In CORBA this glue code is called the "Broker."

dcomplex

Definition: The sum of a real number and an imaginary number is called a complex number. Babel supports complex numbers as a basic type via the basic types "fcomplex" and "dcomplex."
SIDL keyword: dcomplex is a data type built into SIDL. The name is short for "double complex." It stores a complex number via 2 64-bit floating point variables, one for the real part, and one for the imaginary part.

dense

Definition: A dense array is an array where all the dimensions are "densely packed," or, in terms of memory addressing, there are no "spaces" between array elements. For example, if a one-dimensional SIDL array of 10 elements is created, it will be densely packed. However, if a slice of the array is taken with a stride of 2, the resulting array will use the same data as the original array. However, the new array will be only five elements long, and will only consist of the even elements of the original array. This is not densely packed. Example:

Array 1: 0 1 2 3 4 5 6 7 8 9
Array 2: 0 – 2 – 4 – 6 – 8 –

developer

Babel: There are two anticipated user types for Babel, both are kinds of programmers. The person referred to as the "developer" is the person developing a Babelized library. The "user" is the person who writes a program using a Babelized library.

DLL

Definition: Dynamically Linked Library. A type of library that can be linked to dynamically at runtime by passing its name as a string to the dlopen() function.

double

Definition: A double is a 64-bit floating point number.
SIDL keyword: SIDL support double as a basic type.

DTD

Document Type Definition. Defines the grammar of the XML files. http://www.w3.org/2002/xmlspec/

dynamic linking

Definition: The action of dynamically linking to DLLs at runtime.

enum

Definition: Enum is a shortend form of the word enumeration. An enumeration is used to assign numbers to a set of variable names, that is, enumerate the set of variable names.
SIDL keyword: enum is a reserved word in SIDL. It is used for defining enumerations. In Babel, enumerations are a way of binding integer constants to names.

enumeration

In Babel, enumerations are a way of binding integer constants to names. See subsection 5.3.

exception

Definition: The idea of an exception is that if a method encounters a problem it cannot handle, it interrupts its execution and "throws" and exception. Hopefully some function up the call stack will "catch" the exception and know what to do about the problem. It is a useful form of error handing that SIDL supports. Exception is not a reserved word in SIDL (but `throw` is).

extends

OOP concept: See inheritance.
SIDL keyword: extends is a SIDL reserved word. It is used to declare "like-type" inheritance. For example, a class may extend another class, or an interface may extend multiple interfaces, but a class cannot extend an interface, nor can an interface extend a class.

external stubs

When building a Babelized library, its also important to note if your code has dependencies to other Babel types not in your library. These types often appear as base classes, argument types, or even exception types. Your library will need stubs corresponding to all these types, so it is best to put these in your library as well. We call these external stubs. See subsection 16.2.3

external types

External Types are variable or object types that are not defined in the current class. In a class foo.Bar, sidl.Integer, or sidl.BaseClass would be external types.

fcomplex

fcomplex is a data type built into SIDL. The name is short for "float complex." It stores a complex number via 2 32-bit floating point variables, one for the real part, and one for the imaginary part.

final

final is a SIDL reserved word. It is a method modifier. A final method is inherited by subclasses, but its implementation can never be overwritten. It is the "final" version of the implementation.

float

float is a data type built into SIDL. It is a 32-bit floating point number. float is short for floating point.

full name

Overloaded Babelized methods called from non-object oriented languages, such as C and FORTRAN 77, have 2 method names. The full name consists of the concatenation of the package name, class name, method name and type extension. The short name is missing the type extension. See subsection 5.6.

fundamental types

Fundamental types are the basic types that SIDL supports natively. bool, int, char, long, float, double, fcomplex, dcomplex, opaque, and string.

glue

Most of the code that Babel generates is "glue" code. "Glue" code sits between the caller and the implementation to allow communication between them. We use the term glue to refer to the stub, IOR, and skel files.

HTML

Hypertext Markup Language http://www.w3.org/MarkUp/

implementation

In Babel, the implementation is the code placed in the server side Impl files. It is the code that Babel used glue code to allow you to call to.

implements

implements is a SIDL reserved word. It is used when a class inherits from one or more interfaces. However, in this case the word "to implement" is not quite taken seriously. If a class implements an interface it inherits its methods, and may be cast to that interface, but if the programmer actually wished to implement any of the interface methods, he must redeclare them in the SIDL class. Any un-redeclared method is assumed abstract and will not appear in the Impl files. If there are any abstract methods in a class, that class is automatically abstract.

implements-all

implements-all is a SIDL reserved word. It takes the place of "implements." It is used when a class inherits from one or more interfaces, and the programmer definitely wants to write implementation code for each method in the named interfaces. If the programmer uses "implements-all" he does not have to redeclare the interface methods. See Section 5.6

import

import is a SIDL reserved word. It is used to bring other packages into scope. Packages may be accompanied by a version number.

in

in is a SIDL reserved word. Each parameter passed though Babel must be declared as in, out, or inout. Each of these modes has certain rules and implication associated with it. In means "pass this variable by value to the implementation." See Section 5.2.

independent arrays

Independent arrays are arrays that manage their own data. When all the references to an independent are deleted, the array data is garbage collected. The other kind of array is a borrowed array.

inheritance

In normal object-oriented programming, inheritance is the ability of a "super" or "parent" class or interface to pass its characteristics (methods and instance variables) on to its subclasses, allowing subclasses to reuse these characteristics.

Of course, in SIDL we cannot define instance variables, so in SIDL inheritance only refers to method inheritance. In SIDL inheritance is is declared with the reserved words `extends` and `implements`.

inout

inout is a SIDL reserved word. Each parameter passed though Babel must be declared as in, out, or inout. Each of these modes has certain rules and implication associated with it. Inout means "pass this variable by reference to the implementation. The implementation may do whatever it wants with the reference, but it should return something. Possibly a new variable." See Section 5.2.

instance method

An instance method is a method that must be associated with an object instance. These methods probably rely on some state in the instance, so they cannot be divorced from it. In Object Oriented languages, you call these methods on an instance, in Babelized non-OO languages like C, you pass an instance in as the first argument to one of these methods.

int

int is a data type built into SIDL. It is a 32-bit integer variable int is short for integer.

int32_t and int64_t

The ANSI C standard way of declaring an integer that is definitely 32 or 64 bits.

interface

An interface is a declaration of a set of methods with no information given about their implementation. All interface methods are abstract. An interface cannot be instantiated. However, a class may inherit from multiple interfaces. The

purpose of interfaces is to give objects that are conceptually similar but internally different a common interface so that code may treat them the same, or seamlessly exchange them.

interprocess

Interprocess means "between processes." It is normally used to refer to "interprocess communication," where two or more processes find some way to communicate. Interprocess communication is one of the goals of babel with RMI.

IOR

Intermediate Object Representation. IOR code is where Babel does all its work maintaining arrays, Babel objects, reference counting, etc.

JNI

Java Native Interface. The JNI is what allows Java to call to C and C++. It is referred to as calling native code because while Java runs in a virtual machine, but C and C++ run on the real machine, or run "natively."

language interoperability

Language interoperability is Babel's main purpose. Language interoperability technology allows different computer languages to call each other methods and communicate despite problems with calling conventions and differing variable types.

local

A method (or other identifier) is considered local if it is defined or declared in the current class or method. Sometimes a more specific term like, "local to the method" or "local to the class" is used. There is also a SIDL keyword local that modifies methods. If a method is local is can only be called in-process, and cannot be exported over RMI.

long

long is a data type built into SIDL. It is a 64-bit integer variable long is short for long integer. Note: Python sometimes has trouble with longs, see Section 11.7 for more details.

method

Method is the word commonly used in Java for what is called, in some other languages, a function, subroutine, or procedure. Methods are a piece a code that is called by a name. Instance methods depend on an object instance, and are allowed to read and manipulate that objects data. A static method does not depend on an instance, and therefore can only access class data and what data is passed in to the method.

namespace

A namespace is a way of logically divvying up globally accessible names. This helps in avoiding conflicts between globally accessible methods, classes, data, etc. They are mainly a feature of C++.

nonblocking

nonblocking is a SIDL method attribute. A nonblocking method is split into two parts. The invocation, method_send(), makes the call and immediately returns a sidl.rmi.Ticket. Later, the Ticket can be used to check if the method has returned, and retrieve the out arguments if it has with method_recv(). Nonblocking methods are really only useful with RMI where it allows the client to mix computation and communication more freely.

non-strided

A non-strided array is a dense array. See the glossary entry for dense.

Object model

The Object Model is the of rules that regulates the definition, creation, and use of classes and objects in a language. To read about the SIDL object model see Section 5.6

OMG

Object Management Group http://www.omg.org/

oneway

oneway is a SIDL method attribute. A oneway method is guaranteed to have no out arguments at all, it cannot even throw exceptions. This is so it can be invoked by a oneway message on RMI. oneway is really only useful with RMI.

opaque

opaque ia a data type build into SIDL. The word opaque is an adjective meaning "not transparent." In SIDL, an opaque is a 64-bit variable that cannot be touched or modified by the holder. It is normally used to hold pointers that cannot be understood by the current language or in the current context.

out

out is a SIDL reserved word. Each parameter passed though Babel must be declared as in, out, or out. Each of these modes has certain rules and implication associated with it. Out means "pass this (null) variable by reference to the implementation. The implementation is expected to fill the reference with a new variable to be passed back to the client." See Section 5.2.

package

A package is a container and namespace for conceptually linked classes and interfaces. Generally it is good practice to have one package per SIDL file.

pass-by-copy

Pass-by-copy referes to one of the two major ways arguments are passed to methods (the other is pass-by-reference). In a pass-by-copy scheme, arguments are always copied when they are passed, so that changeing the value of argument in the callee does not effect the value of the caller's variable. This is particularly important to Babel RMI, where object can be passed either by copy or reference.

pass-by-reference

Pass-by-reference referes to one of the two major ways arguments are passed to methods (the other is pass-by-copy). In a pass-by-reference scheme, arguments remain in their original memory localtion and a pointer to them is passed to the callee method. This means that if the callee changes the value of an argument, the value of the caller's variable changes as well. This is particularly important to Babel RMI, where object can be passed either by copy or reference.

pass-by-value

See pass-by-copy

PIC

Position Independent Code is for making dynamically loadable libraries. PIC contains and extra level of indirection to allow the correct methods to be found dynamically at runtime.

preprocessing

Code preprocessing is a step, prior to compilation, where various simple, automatic code modifications are made. For example, int C, #include files are included, and #define macros are textually duplicated throughout the code. In some cases, such as Babel FORTRAN 90, method names are "mangled" to reduce their size under the method name character limit.

private data

Private data is data that is only accessible locally, inside an object. In Babel, all Babel object data is private and cannot be accessed by other SIDL objects.

process

A process is a running program that exists in its own memory space and can therefore run in parallel with other processes.

protocol

A protocol is formal description of message formats and the rules that two computers must follow in order to exchange messages.

Babel RMI may use any protocol that implements the Babel RMI API. This API is defined in sidl.io.Serializer and sidl.io.Deserializer.

reference counting

Reference counting is the form of garbage collection used in Babel. Each object keeps a "reference count." When that count reaches zero, the object is destroyed and the memory reclaimed. In some languages teh counting is handled automatically, in some, like C, the developer must explicitly add and subtract from the reference count. (Using the functions addRef and deleteRef.) The internal implementation of deleteRef literally has an if statement that says "If the count is 0, free this memory," so if the reference count of an object goes below one, all references to the object are immediately invalid.

Remote Method Invocation

Remote Method Invocation (RMI) is Object Oriented Remote Procedure Call (RPC). Where RPC allows a user to call procedures on remote machines, RMI allows the user to call methods on objects that may or may not exist on a remote machine. This has the advantage of being more natural and makes local and remote object interchangeable.

reverse engineering

Reverse Engineering is the practice of inspecting the behavior of an existing program to understand more about how it works. Babel does not support this, or any forms of inspecting or modifying compiled code.

RMI

See Remote Method Invocation

RPC

See Remote Method Invocation

serialization

Serialization is a process to encode a data structure as a sequence of bytes. This is the method used by most object oriented system to save objects to files or pass objects over a network connection. Babel RMI uses serialization to pass objects by copy over the network.

shared library

A shared library is a set of methods that may be used by multiple different programs without recompilation of the library.

short name

Overloaded Babelized methods called from non-object oriented languages, such as C and FORTRAN 77, have 2 method names. The full name consists of the concatenation of the package name, class name, method name and type extension. The short name is missing the type extension. See subsection 5.6.

SIDL

Scientific Interface Definition Language. The language used by Babel to describe how Babel glue code should be generated. See Chapter 5.

single process

A single process program is a program that only uses one process to complete its work. One of the features of Babel is that it is able to facilitate language interoperability in a single process, which saves the extra overhead of interprocess communication.

skeleton

The Babel skeleton code is the opposite of the Babel stub code. The Stub code facilitates the method call from client to IOR, and the skeleton code facilitates the method call from IOR to implementation.

SO

Shared Object. A Unix catch all term for shared and dynamically loadable libraries.

SPMD

Single Program Multiple Data. The term used to describe parallel programs that use multiple processes running the same code working on different data to solve a problem.

state (of an object)

Object state refers to the data that an object holds. For example, if an object holds one integer, that integer holds the objects state. It is assumed that instance methods modify or use an object's state in some way. If a method does not use the object state in any way, it should probably be a static method.

static

A static method is a method that does not depend on an object instance to run. It should have no need of any data of any particular object, it should only depend on the data that is passed into it. As such, unlike instance methods, it does not need to run on an instance of the class it is associated with. In Babelized C, this means the first argument to the function is not an object instance. In Java, this means the function not called on an object, but referenced by the class name.

static linking

Static linking refers to the practice of linking code at compile time, rather than dynamically at runtime. It has a speed advantage over dynamically linked code, but lack flexibility.

string

string is a data type built into SIDL. It stores a set of characters. It has no predefined length.

stub

The Babel stub code is the opposite of the Babel skeleton code. The Stub code facilitates the method call from client to IOR, and the skeleton code facilitates the method call from IOR to implementation.

SWIG

Simplified Wrapper and Interface Generator http://www.swig.org/ SWIG is a language interoperability tool that is not IDL based, but has certain other drawbacks.

tarball

Tarball is a common way to refer to a set of directories and files organized into a single file using the Unix tar command. It is often gzipped.

throws

throws is a SIDL reserved word. It is used the tell SIDL that a method may throw the named SIDL exception, and code should be generated to pass it to the client.

type

A type describes what sort of information a variable stores, and usually how much space that information takes up. Classes and interfaces are user defined types, there are also fundamental types like int and bool.

URL

Uniform Resource Locater. Often thought of as a pointer to a web resource.

user

There are two anticipated user types for Babel, both are kinds of programmers. The person referred to as the "developer" is the person developing a Babelized library. The "user" is the person who writes a program using a Babelized library.

version

version is a reserved word in Babel that is used to declare a version for a given package, or to declare what version of a given package should be used.

virtual

Virtual is the opposite of final. All SIDL methods are virtual by default. A virtual method is a method that may be overridden in subclasses.

VM

Virtual Machine

void

a reserved word in Babel, used to state that a function has no return type.

VPATH

If you want to build software in a separate directory from where the tarball was untarred, this is called a "VPATH build". VPATH builds are useful if you want to build Babel multiple times with various compilers, flags, or you have a shared file system across multiple platforms. It separates the code you generate from things that you were given.

XML

Extensible Markup Language. http://www.w3.org/XML/ A standardized data exchange format.

Bibliography

[1] Babel homepage. `http://www.llnl.gov/CASC/components/babel.html`.

[2] David E. Bernholdt, Wael R. Elwasif, James A. Kohl, and Thomas G. W. Epperly. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages (POHLL-02)*, New York, NY, June 2002.

[3] CCAFE homepage. `http://www.cca-forum.org/~baallan/ccafe`.

[4] Bradford Cobb, Gary Hook, Christopher Strauss, Ashok Ambati, Anita Govindjee, Wayne Huang, and Vandana Kumar. AIX linking and loading mechanisms. `http://www-1.ibm.com/servers/esdd/pdfs/aix_11.pdf`, May 2001.

[5] Common Component Architecture (CCA) Forum homepage. `http://www.cca-forum.org`.

[6] Tammy Dahlgren, Tom Epperly, and Gary Kumfert. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, version 0.8.4 edition, April 2003.

[7] Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

[8] Eric Eide, Jay Lepreau, and James L. Simister. Flexible and optimized IDL compilation for distributed applications. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 1998.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, July 1996. Available at `http://java.sun.com`.

[10] Michi Hennig and Steve Vinoski. *Advanced CORBA Programming with C++*. Professional Computing. Addison-Wesley, 1999.

[11] International Organization for Standardization, Geneva. *ISO/IEC 14882 Standard for the C++ Programming Language*, 1998.

[12] Bill Janssen, Mike Spreitzer, Dan Larner, and Chris Jacobi. *ILU Reference Manual*. Xerox Corporation, November 1997. Available at `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

[13] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 2001.

[14] Scott Meyers. *More Effective C++: 35 New Ways to Improve your Programs and Designs*. Professional Computing. Addision-Wesley, 1996.

[15] Scott Meyers. *Effective C++: 50 Specific Ways to Improve your Programs and Designs*. Professional Computing. Addision-Wesley, 2 edition, 1998.

[16] Microsoft Corporation. *Component Object Model Specification (Version 0.9)*, October 1995. See `http://www.microsoft.com/oledev/olecom/title.html`.

[17] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1998. Available at `http://www.omg.org/corba`.

[18] SciDAC: Scientific Discovery through Advanced Computing. `http://www.science.doe.gov/scidac`.

[19] SCIRun homepage. `http://www.sci.utah.edu`.

[20] John Shirley, Wei Hu, and David Magid. *Guide to Writing DCE Applications*. O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

[21] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

[22] U. S. Department of Energy (DOE) homepage. `http://www.energy.gov`.

[23] Norm Walsh. *DocBook*. O'Reilly, 2000.

[24] XCAT homepage. `http://www.extreme.indiana.edu/xcat`.

Index