ILU 2.0alpha14 Reference Manual

Bill Janssen <janssen@parc.xerox.com>
Mike Spreitzer <spreitzer.parc@xerox.com>
Dan Larner <larner.parc@xerox.com>
Chris Jacobi <jacobi.parc@xerox.com>

(with contributions from Doug Cutting, Frank Halasz, and Denis Seversen)

(typeset 21 January 1999)

This document describes version 2.0alpha14 of the Inter-Language Unification (ILU) system.

Lots of people contributed significant amounts of code to the ILU system, including (alphabetically): Joachim Achtzehnter, Judy Anderson, Antony Courtney, Doug Cutting, Mark Davidson, Ken Fishkin, Frank Halasz, Scott Hassan, Rob Head, Chris Jacobi, Bill Janssen, Swen Johnson, Dan Larner, Martin von Loewis, Bill Nell, Paula Newman, Ansgar Rademacher, Dennis Seversen, Bridget Spitznagel, Mike Spreitzer, Owen Taylor, Farrell Wymore, and Rick Yardumian.

Many others have contributed in other ways, including our reviewers, alpha and beta testers, and regular users. The list includes (but is not limited to): Shridhar Acharya, Joachim Achtzehnter, Judy Anderson, Maria Perez Ayo, Mike Beasley, Erik Bennett, Dan Brotsky, David Brownell, Bruce Cameron, George Carrette, Philip Chou, Daniel W. Connolly, Antony Courtney, Doug Cutting, Mark Davidson, Jim Davis, Larry Edelstein, Paul Everitt, Bill Fenner, Josef Fink, Jeanette Figueroa, James Flagg, Steve Freeman, Mark Friedman, Jim Gettys, Gabriel Sanchez Gutierrez, Jun Hamano, Bruno Haible, Scott W. Hassan, Carl Hauser, Rob Head, Michi Henning, Andrew Herbert, Angie Hinrichs, Ben Hurwitz, Roberto Invernici, Christian Jacobi, Swen Johnson, Gabor Karsai, Nobu Katayama, Dan 'Bud' Keith, Sangkyun Kim, Ted Kim, Don Kimber, Steve Kirsch, Dan Larner, Carsten Malischewski, Larry Masinter, Fernando D. Mato Mira, Fazal Majid, Steven D. Majewski, Fernando D. Mato Mira, Michael McIlrath, Scott Minneman, Masashige Mizuyama, Curtis McKelvey, Chet Murthy, Farshad Nayeri, Bill Nell, Les Niles, T. Owen O'Malley, Annrai O'Toole, Andreas Paepcke, Jan Pedersen, Karin Petersen, Steve Putz, George Robertson, Joerg Schreck, Ian Smith, Bridget Spitznagel, Peter Swain, Marvin Theimer, Lindsay Todd, P. B. Tune, Bill Tutt, Kevin Tyson, Bill van Melle, Guido van Rossum, Brent Welch, Jody Winston, Rick Yardumian.

1 ILU Concepts

1.1 What ILU Does

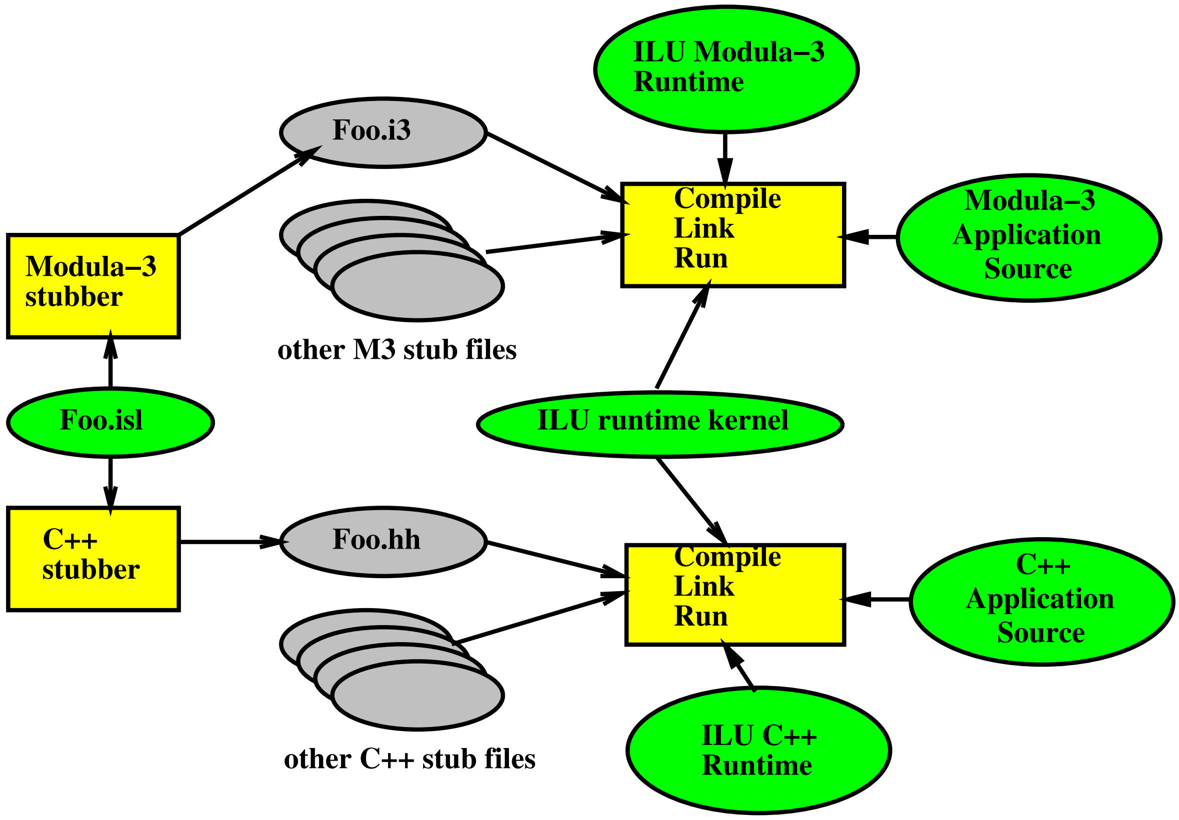
ILU is primarily about interfaces between units of program structure; we call these units *modules*. The notion is that each module enscapsulates some logical part of a program, that has high 'cohesiveness' internally, and low 'coupling' to other parts of the program. ILU provides you with a way of writing down an object-oriented *interface* to the module; that is, a set of object types and other types, constants, and exceptions that another module would use to communicate with it. This interface can then be processed by various ILU tools to implement that communication.

ILU allows many different binding relationships between modules. The modules can be parts of one program instance, all written in the same language; they can be parts written in different languages, sharing runtime support in one memory image; they can be parts running in different program instances on different machines (on different sides of the planet). A module could even be a distributed system implemented by many program instances on many machines. A particular module might be part of several different program instances at the same time. ILU does all the translating and communicating necessary to use all these kinds of modules in a single program. It optimizes calls across module interfaces to involve only as much mechanism as necessary for the calling and called modules to interact. In particular, when the two modules are in the same memory image and use the same data representations, the calls are direct local procedure calls --- no stubs or other RPC mechanisms are involved. The notion of a 'module' should not be confused with the independent concept of a *program instance*; by which we mean the combination of code and data running in one memory image. A UNIX process is (modulo the possibilities introduced by the ability, in some UNIX sytems, to share memory between processes) an example of a program instance.

Because ILU standardizes many of the issues involved in providing proper inter-module independence, such as memory management and error detection and recovery strategies, it can be used to build language-independent *class libraries*, collections of re-usable object definitions and implementations. Because one of the design goals of ILU was to use existing standards for various pieces, rather than inventing anything new, ILU can be used to implement ONC RPC or Xerox Courier services, or clients for existing ONC RPC or Xerox Courier services. ILU also includes an implementation of the Object Management Group's CORBA Internet Inter-Orb Protocol (*IIOP*), and can be used to write CORBA services or clients, as well.

1.2 How ILU Works

The approach used by ILU is one common to standard RPC systems such as Sun's ONC RPC, Xerox's Courier, and most implementations of OMG's CORBA. An interface is described once in some 'language-neutral' interface specification language. Types and exceptions are described; exported functionality is specified by defining *methods* on object types. Tools are then run against the interface description to produce *stubs* for particular programming languages; these stubs can bind to, call, and be called from stubs generated from the same interface description for a different programming language. The stub code is then linked with the application code, some language-specific code containing any necessary ILU support for that programming language, and the *ILU kernel library*, which is code written in ANSI C. The following diagram illustrates the process:

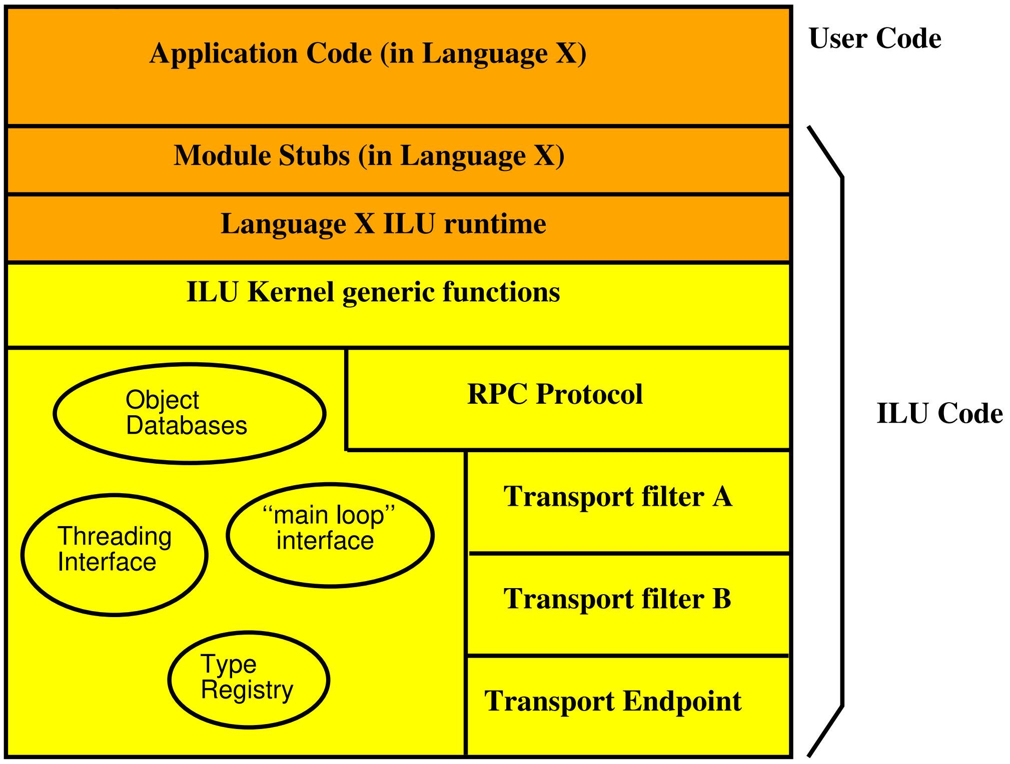


Two ILU modules in different languages

and different address spaces

Several modules may be linked together, for a standalone use. ILU stubs are generated in such a way that applications which link a caller and callee written in the same language directly together suffer no calling overhead. This makes ILU useful for defining interfaces between modules even in programs that do not use RPC.

Different modules of the program may be written in different programming languages. These can either be linked together in the same address space, if the runtimes of the different languages allow that, or they can be used to make separate network servers and clients. In the case of a network service, the memory layout for the program would be something like



An ILU address space

1.3 Core ILU Concepts

1.3.1 Objects

ILU is *object-oriented*. By this, we mean that object types serve as the primary encapsulation mechanism in ILU. All functionality is exported from a module as methods that can be invoked on an instance of some object type, rather than as simple procedures. The object instance provides the context within which methods are executed. The object type system provides subtyping ('inheritance' of interfaces (ILU does not address object implementation)), to aid in structuring of interfaces.

With respect to a particular ILU object instance, a module is called the *server* if it implements the methods of that object, or a *client* if it calls, but does not implement, the methods of that object. One module can thus be a client of one object, and the server of another. An ILU object can be passed as a parameter to or result of a method call, and can be (in) the parameter to an exception. An object may be passed from its server to a client, from a client to its server, or between two clients, in any of the above three kinds of position. Unlike some RPC systems, there can be multiple ILU objects of the same type, even on one machine, even within one program instance.

For a given ILU object, there will, in general, be multiple *language-specific objects*; each is an ''object'' in one of the programming languages used in the system. One language-specific object, designated the *true object*, actually provides the implementation of the ILU object; it is thus part of the server module. The true object's methods are written by the programmer, not generated by ILU. The other language-specific objects are *surrogate objects*; their methods are actually RPC stubs (generated by ILU) that call on the true object. A surrogate object is used by a client module when the server module is in a different program instance or uses different data representations.

1.3.1.1 Kernel Servers

Each instance in an ILU address space is associated with a *kernel server*, a construct which manages a group of objects. Kernel servers are found in both client and server modules. Each kernel server has a *server ID*, a universally unique string ID. The server ID makes up part of the object ID of the instances supported by the server. Some kernel servers contain only surrogate instances, and are called *surrogate servers*; others contain both surrogate and true instances, and are called *true servers*.

Kernel servers serve as the locus of communication between two address spaces. A true server may have a number of *ports* associated with it; a port is a mechanism by which other address spaces can interact with objects in this address space. Other address spaces use the port by creating a surrogate server which mirrors the true server, and opening a *connection* from the surrogate server to the true server. Calls from surrogate instances on true instances are carried along this connection. A true server may have multiple ports, each of which may provide connectability via different RPC protocols or transport mechanisms.

1.3.1.2 Subtyping (interface inheritance)

The object model specified here provides for multiple interface inheritance. It is intended that the subtype provide all the methods described by its supertypes, plus possibly other methods described directly in the subtype description. It is expected that in languages which support multiple-inheritance object models, that an ILU inheritance tree will be reflected in the language-specific inheritance tree. In a single-inheritance language, or a non-object-oriented one, an ILU-specific multiple-(interface-)inheritance object system must be embedded.

1.3.1.3 Subtype Relationships

In the ILU type system, the only subtyping questions that arise are between two object types. This is because ILU employs only those OOP features common to all languages supported.

Subtyping in ILU is based on structure and name; we include the names in the structure, and thus need only talk about structure. An object type declaration of the form defined later constructs a structure of the form

(OBJTYPE
  SINGLETON: *singleton-protocol-info*
  OPTIONAL: `Boolean`
  COLLECTIBLE: `Boolean`
  AUTHENTICATION: *authentication-type*
  SUPERTYPES: *supertype-structure, ...*
  METHODS: *method-structure, ...*

LEVEL-BRANDS: (*interface-name*, *interface-brand*,
type-name, *type-brand*))

Structure A is a subtype of structure B iff either (1) A and B are equal structures, or (2) one member of A's *supertype-structure*s is a subtype of B.

Note that the level-brands include the interface name and (optional) brand, as well as the name and (optional) brand of the type being declared. Thus, two declarations of subtypes of the same type normally create distinct subtypes, because they would declare types of different names, or in interfaces with different names. When the interface name and the type name are the same, this does not cause a distinction, although other structural differences might. If the programmer wants to indicate that there's a semantic distinction, even though it doesn't otherwise show up in the structure, s/he can use different interface brands and/or different type brands. These distinctions can be made between declarations in different files, or between successive versions of a declaration in a file that's been edited.

1.3.1.4 Singleton Object Types

Many existing RPC protocols and servers do not have the notion of multiple instances of a type co-existing at the same server, so cannot use the instance discrimination information passed in ILU procedure calls. To support the use of these protocols and servers, we introduce the notion of a *singleton* object type, of which there is only one instance (of each singleton type) at a kernel server. Note that because a single address space may support multiple kernel servers, this means that in a single address space, there may be multiple instances of the same singleton type. When a method is being called on an instance of a singleton type, no instance discrimination information is passed. Singleton types may not be subclassed.

1.3.1.5 Instantiation

To use (e.g., call the methods of) an ILU object, a client must first obtain a language-specific object for that ILU object. This can be done in one of two ways: (1) the client can call on a language-specific object of a different ILU object to return the object in question (or receive the object in a call made on the client, or in the parameter of an exception caught and handled by the client); or (2) certain standard facilities can be used to acquire a language-specific object given either addressing or naming information about the ILU object. The addressing information is called a *string binding handle* (SBH), and the ILU runtime library includes a procedure to acquire a language-specific object given a string binding handle for an ILU object (in strongly-typed languages, this procedure is typed to return an object of the base type common to all ILU objects in that language).

Every creation of a surrogate instance implies communication with the server module, and binding of the surrogate instance to the true instance. ILU may attempt to perform this communication when it is actually necessary, rather than immediately on surrogate instance creation.

The process of creating an instance may bootstrapped via a *name service*, such as the PARC Name-and-Maintenance-Server (`NMS`), which allows servers to register instances on a net-wide basis. A server registers a mapping from naming information to a string binding handle. The client-side stubs for an interface include a procedure that takes naming information, looks up the corresponding string binding handle in the name service, and calls the above-mentioned library routine to map the SBH to a language-specific object. Alternatively, a client can do those steps itself, using an ILU runtime library procedure to acquire a language-specific object for the name service.

1.3.1.6 String Binding Handle

In ILU, there is a string-based representation for a reference to an object. That representation consists of a single string, called a *string binding handle*. ILU uses string binding handles when marshalling object references for RPC. ILU also allows applications to interconvert between objects and string binding handles. This is necessary when dealing with name services, and useful in other circumstances.

A string binding handle contains several different pieces of information:

- The *server ID*, a string which identifies the particular *kernel server* that the object belongs to; any program can separate its objects into one or more groups, each group associated with a different kernel server. Two objects from the same kernel server are called *sibling* objects.
- The *instance handle*, identifies which object on the particular kernel server is being specified.
- The *most specific type ID* (also called the *MSTID*), a type fingerprint for the most specific type of the object.
- The *contact info*, specifies one or more of the ways by which a client of the object can communicate with it.

The server ID, instance handle, and MSTID may each contain any ASCII character other than NUL. They are composed into the string binding handle according the the IETF rules for URLs, but the precise form of the URL is not specified here. (In versions of ILU before 2.0, string binding handles had a completely different syntax.)

The pair `(`*server ID*`,` *instance handle*`)` are also known as the *object ID* (or *OID*) of the object, because together they form a universally unique ID for the object.

The *contact info* part contains one or more contact info sequences, each describing one particular way of communicating with the object's kernel server. Each contact info sequence consists of a series of fields. The first field is known as the *protocol info*, and names a particular RPC protocol, and any parameters that might influence the way in which this protocol would be used. Each of the succeeding fields specifies *transport info*, which defines a way of transforming or communicating data, and any parameters which might influence that transport method. There may be many sequences of contact info in any one string binding handle (but ILU currently ignores all but the first).

1.3.1.7 Siblings

Some ILU object instances may have implementation dependencies on private communication with other instances. For example, imagine an object type `time-share-system`, which provides the method

`ListUsers()`, which returns a list of ‘‘user’’ instances. Imagine that `time-share-system` also provides the method `SetUserPriority(u : user, priority : integer)`. We would like to be able to provide some assurance that the user instance used as a parameter to `SetUserPriority` is an instance returned from a call to `ListUsers` on the same instance of a `time-share-system`, because the way in which `SetUserPriority` is implemented relies on the user being a user of that particular `time-share-system`.

The ILU model provides the notion of a *sibling object*. Two instances are siblings if their methods are handled by the same kernel server. Instances that are non-discriminator parameters to methods may be specified in ISL as having to be siblings of the discriminator.

1.3.1.8 Object Tables (or, Just-in-Time Objects)

True objects may either be created explicitly, or upon arrival of calls on them. The second option is exercised via a feature currently called *object tables* (from "hash tables", since they map a string, the instance handle, to an object -- "object factories" might be a less surprising term). After the object table creates an object, the server module then continues to manage the object's existence --- in the same way(s) it manages other objects it creates. This means a server need not hold in memory all of its objects at once, which may be quite important.

A true kernel server may optionally include an object table, whose job is to map an instance handle (see Section 1.3.1.6 [String Binding Handle], page 7) to the object it identifies. ILU's runtime will consult the object table when a call is received for an object not currently reified. The object table can either explicitly create the named object, or refuse (thus declaring the instance handle invalid).

This mapping operation is invoked with certain of the ILU runtime's mutexes (see Section 12.1.4 [Thread Synchronization], page 193) held, because it is an extension of a delicate part of that runtime. The server's mutex is held in all cases, and the global mutex "gcmu" is also held if the resulting object is expected to be of a `COLLECTIBLE` type. The fact that these mutexes are held restricts what an application can do inside this mapping procedure.

1.3.1.9 Server Relocation

It is sometimes useful to have a ‘dummy’ true kernel server, that will redirect any requests to it to a real true kernel server somewhere else. This can be used for load balancing, automatic start-up of services, implementation of a redirecting name service, code migration, and other various purposes. ILU supports this via a mechanism called *server relocation*. This mechanism allows a function to be associated with a true kernel server, which is called when a request arrives at that kernel server over a connection which uses a *relocating protocol*. A relocating protocol is a protocol that carries relocation requests, such as the CORBA `IIOP`, or the HTTP-NG `w3ng` protocol. The relocation function returns new contact info for the kernel server, which is sent back to the caller. The caller then closes the existing connection and opens a new connection according to the specified contact info.

1.3.2 Garbage Collection

A simple form of *garbage collection* is defined for ILU objects. If an object type is tagged as being collectible, a server that implements objects of that type expects clients holding surrogate instances to register with it, passing an instance of a callback object. When a client finishes with the surrogate, the client unregisters itself. Thus the server may maintain a list of clients that hold surrogate instances. If no client is registered for an object, and the object has been dormant (had no methods called on it) for a period of time *T1*, the server may feel free to garbage collect the instance. *T1* is determined by human concerns, not network performance: *T1* is set long enough to allow useful debugging of a client.

To deal with possible failure of a client process, we introduce another time-out parameter. If an instance with registered clients has been dormant for a period of time *T2*, the server uses the callback instance associated with each client to see if the client still exists. If the client cannot be contacted for the callback, the server may remove it from the list of registered clients for that instance.

If a client calls a method on a surrogate instance of a true instance which has been garbage-collected (typically because of partitioning), it will receive the `ilu.ProtocolError` exception, with detail code `ilu.NoSuchInstanceAtServer`.

1.3.3 Connections

ILU does not (directly) expose to the application programmer any notion of ‘‘connections’’. That is, the called module has no pointer back to the caller, and no notion of how to do anything with the caller aside from returning a result message. Credentials passed in the request message can identify the caller, but not necessarily the location the call is made from. Protocols that need such information should pass it explicitly as an argument (an instance of an object type with methods defined on it) to the method.

1.3.4 Pipelining

ILU’s mechanisms avoid introducing blocking into a distributed program. This is because ILU does not try to track the identity of a thread of execution as it crosses program boundaries. So if ILU were to make one call wait for the completion of another, this would be a potential cause of deadlock.

It is possible for the programmer to explicitly inform ILU that one call’s execution is not necessary for the completion of another. This is done indirectly, via a concept called a *pipeline*. A client can create a pipeline (any number, actually), and associate any collection of its calls with a pipeline (at most one pipeline per call). Making such associations asserts to ILU that none of the calls is needed for any other of them to complete. This allows ILU to block some of them until others complete.

Which will be blocked, and why would a client want to do this to itself? The answer has to do with connections. You remember, those things the previous section says are not exposed to applications. It’s true that they’re not directly exposed. But we’ll admit here that they exist, and consume resources. Sometimes it’s important to minimize those resources. When using a non-concurrent RPC protocol, ILU avoids introducing blocking by opening as many parallel connections as the client has concurrent calls to the same server. Some clients would prefer that their concurrent calls block instead of consume multiple connections. Such clients can use pipelines to enable this behavior.

1.3.5 Call Order Preservation

ILU does not normally guarantee that the server application will receive calls in the same order that the client makes them (of course, ILU doesn't promise to violate causality --- it just doesn't do any work to give you anything more). This is a particularly interesting issue when making a series of asynchronous calls (because there are no replies to carry causality). You might think that when using a transport, such as TCP, that guarantees ordering, call order preservation will follow as a consequence. But it's not that simple (i.e., ILU may use multiple connections in parallel and series, and TCP provides no ordering guarantees between connections).

However, it's possible for a client application to explicitly request a guarantee of call order preservation for a given collection of its calls. This is done indirectly through an object called a *serializer*. A serializer represents an instance of the serialization guarantee. This guarantee is with respect to a particular server and collection of calls. It guarantees that those calls will be received by the server application in the same order as they were made by the client application --- except that client calls that return after a *barrier call* may be received before client calls that return before that same barrier call. A barrier call is one that raises the BARRIER exception, which is an ILU-specific system exception. Remember that ASYNCHRONOUS calls *do* return, they just do so particularly quickly.

Special considerations apply when these calls are issued concurrently. Two calls are considered to have been issued concurrently if each call is initiated before the other returns. In a multi-threaded runtime, they client may issue concurrent calls under the same instance of the serialization guarantee, and the ILU runtime will put them in some serial order. Note that for two concurrently issued calls, either: (a) the one put first is ASYNCHRONOUS, (b) they both are in the same pipeline, or (c) the one put second is delayed until the one put first returns. In a single-threaded runtime, the client may issue two calls "concurrently" (taking advantage of a nested main loop), but both will execute successfully only if the client is lucky; otherwise, the second one will raise the system exception BAD_PARAM with minor code ilu_bpm_serialConcurrent. Furthermore, when single-threaded, issuing concurrent calls under the same instance of the serialization guarantee but different pipelines will also cause some to raise BAD_PARAM/serialConcurrent.

A client can create any number of serializers, and associate each one of its calls with at most one serializer. This guarantee is only available for servers exported over non-concurrent RPC protocols and reliable transports. Due to current implementation limitations, the default port of the server must satisfy the protocol and transport restriction. If that port does not meet the protocol restriction, serialized calls will fail with the system exception INV_OBJREF with a minor code of ilu_iom_conc_serial (where no other error is noticed first)

1.3.6 Batching

In ILU, a call between address spaces involves sending an *call message* from the caller to the callee. The call message is usually sent immediately upon initiation of the call. However, there is a way for these call messages to be delayed and gathered into *batches* under application control. An application specifies this by use of a meta-object called a *batcher*. A given call may optionally be associated with a batcher, and a batcher may use either or both of two ways to specify when delivery of its buffered call messages should be initiated. The first method is by explicit application call to *push* the batcher. The second is by timeout: a call message's delivery is initiated at most some time constant past the time when composition of the call message completed. Which of these two ways are applicable is specified when the batcher is created, as is the timeout value (if any). Note that we speak here only of *initiation* of delivery, not receipt by any particular layer of the receiver. This feature involves only client-side mechanism, and so may be used with non-ILU servers.

1.3.7 Simple Binding

ILU includes a simple binding/naming facility. It allows a module to *publish* an object, so that another module can import that object knowing only its object ID (as defined in Chapter 1 [ILU Concepts], page 2). It is essentially just a way of binding a URN (the object's ID) to a URL (the object's string binding handle). The interface to this facility is deliberately quite simple; one reason is to allow various implementations.

The interface consists of three operations: *Publish*, *Withdraw*, and *Lookup*. `Publish` takes one argument, an ILU object. `Publish` returns a string that is needed to successfully invoke `Withdraw`. `Withdraw` undoes the effects of `Publish`, and takes two arguments: (1) the object in question, and (2) the string returned from `Publish`. In some langauge mappings, the string is not explicitly passed, but conveyed in the language mapping's representation of ILU objects. `Lookup` takes two arguments: an object ID and a type the identified object should have. If the object with that ID is currently being published, and has the given type (among others), `Lookup` returns that object.

The implementation of simple binding shipped with ILU can use either an ILU service, or a shared filesystem directory, to store information on the currently published objects. This choice must be specified at system configuration time. If the shared filesystem approach is used, this directory must be available by the same name, on all machines which wish to interoperate. The way in which clients interact with binding is the same, regardless of which approach is selected. See Section 11.1 [Binding Names in ILU], page 187 for more information on these implementations.

1.3.8 Error Signalling

ILU uses the notion of an *exception* to signal errors between modules. An exception is a way of passing control outside the normal flow of control. It is typically used for handling of errors. The routine which detects the error signals an exception, which is caught by some error-handling mechanism. The exception type supported in ILU is a termination-model exception, in which the calling stack is unrolled back to the frame which defined the exception handler. Exceptions are signalled and caught using the native exception mechanisms for the servers and clients. A raised exception may carry a single parameter value, which is typed.

1.4 ILU and OMG CORBA

The type and exception model used by ILU is quite similar to that used by the Object Management Group's Common Object Request Broker Architecture (CORBA). We have in fact changed ILU in some ways to more closely match CORBA. Our tools will optionally parse the OMG's Interface Definition Language (OMG IDL) as well as ILU's ISL.

ILU also attempts to address issues that are already upon us, but are not addressed in CORBA 2.0, particularly a uniform way of indicating optional values, and distributed garbage collection.

ILU provides two different interface definition languages, OMG IDL and ILU ISL to enhance portability of ILU modules. The OMG IDL subset understood by ILU is a strict subset of OMG IDL; this means that any ILU modules developed using OMG IDL interfaces should be interoperable with any other CORBA system. Any non-CORBA extensions may only be expressed in ILU ISL, so that any modules which use these extensions must use ILU ISL to express their interfaces, thereby underlining the fact that these modules are not CORBA-compliant. We feel that this dual-interface-language approach will tend to enhance both portability and CORBA-compliance of ILU modules.

ILU does not yet provide some of the features required by a full CORBA implementation. Notably it does not provide a Dynamic Invocation Interface or Dynamic Server Interface, or implementations of either Interface Repository or Implementation Repository. It does not provide the Basic Object Adapter interface, either, but does provide an object adapter with most of the BOA's capabilities, except for those connected with the Interface Repository and/or Implementation Repository.

A number of concepts in CORBA that seem to require further thought are not yet directly supported in ILU: the use of `#include` (ILU uses a more limited notion of ''import''); the notion of using an IDL ''interface'' as both an object type and a name space (this seems to be a ''tramp idea'' from the language C++; in ILU the ''interface'' defines a name space, and the object type defines a type); the notion that all BOA objects are persistent (in ILU, the question of whether an object is persistent is left up to that object's implementation); the notion that type definitions can exist outside the scope of any module or namespace (in ILU, all definitions occur in some interface). Currently, there is no support in ILU for CORBA `contexts`.

2 Defining Interfaces

Module interfaces may be defined in either the Object Management Group's CORBA OMG IDL, or in ILU's native Interface Specification Language (aka ISL). This document describes the syntax and semantics of ISL, and how to translate OMG IDL interfaces into ISL; see *CORBA 2.0* for a specification of the syntax and semantics of OMG IDL.

2.1 General Syntax of ISL

The conventional file suffix for ISL files is '`.isl`'. Some of the ILU tools rely on the name of the file being the same as the name of the interface defined in it, and rely on having only one interface defined in each '`.isl`' file.

An ISL interface contains four kinds of statements: the interface header, type declarations, exception declarations, and constant declarations. Each statement is terminated with a semi-colon.

Many statements in ISL contain lists: lists of the fields in a record, the types in a union, the methods in an object type. All lists in ISL are terminated with an `END` keyword, and the items in the list are separated by commas.

Comments may be placed in an ISL file. They are introduced with the character sequence `(*`, and terminated with `*)`. Comments nest.

2.1.1 Identifiers

All identifiers that appear in ISL are alphanumeric, begin with an alphabetic character, and may contain hyphens.[1] Differences in case are not sufficient to distinguish between two identifiers; however, the case of an identifier may be preserved in its mapping to a specific programming language.

All ILU type names, exception names, and constant names have two parts, an interface identifier and a local identifier. When writing the full name, the interface identifier comes first, followed by a period, followed by the local identifier. If the interface identifier is ommitted in a name, it defaults to the interface identifier of the most recently encountered interface header.

Interface names, type names, exception names, and constant names occur in different name spaces. Thus is is possible to have a type and an exception with the same name.[2]

---

[1] We might forbid two consecutive hyphens or add other restrictions.

[2] We may change this.

2.1.2 Reserved Words

The following words are reserved words in ISL: `[ILU]ALIASED`, `ALLOWS`, `ARRAY`, `ASYNCHRONOUS`, `AUTHENTICATION`, `BOOLEAN`, `BRAND`, `BYTE`, `CARDINAL`, `CHARACTER`, `[ILU]CHARSET`, `CLASS`, `COLLECTIBLE`, `CONSTANT`, `DEFAULT`, `[ILU]DENOMINATOR`, `[ILU]DIRECTIVE`, `[ILU]DIRECTIVE-EXPERIMENTAL`, `DOCUMENTATION`, `END`, `ENUMERATION`, `EXCEPTION`, `EXTENSIBLE`, `FALSE`, `[ILU]FIXED`, `[ILU]FIXEDPOINT`, `FROM`, `FUNCTIONAL`, `ILUSTRING`, `IMPORTS`, `IN`, `INOUT`, `INTEGER`, `INTERFACE`, `LATIN1-CHARSET`, `[ILU]LANGUAGE`, `LIMIT`, `[ILU]LOCAL`, `LONG`, `METHODS`, `MAX-NUMERATOR`, `MIN-NUMERATOR`, `OBJECT`, `OF`, `OPTIONAL`, `OTHERS`, `OUT`, `PICKLE`, `RAISES`, `REAL`, `RECORD`, `[ILU]REFERENCE`, `[ILU]SEALED`, `SEQUENCE`, `SHORT`, `SIBLING`, `SINGLETON`, `SINK`, `SOURCE`, `[ILU]STATE`, `[ILU]STRING`, `SUPERCLASS`, `SUPERCLASSES`, `SUPERTYPES`, `TRUE`, `TYPE`, `TYPEID`, `UNICODE-CHARSET`, `UNION`.

Those words prefixed with `[ILU]` will begin with the three characters `ILU` unless the configuration option --*enable-new-keywords-plain* has been specified in configuring your ILU distribution, in which case the `ILU` prefix will be omitted.

Reserved words may be used as identifiers, by placing them in double quotes, but may not be used as identifiers without quoting.

Other identifiers are worth avoiding, as they may cause problems with specific language implementations. The identifier `t` or `T`, for instance, causes problems with Common Lisp. Language-specific mappings of ISL should try to avoid these problems.

2.2 Statement Syntax

2.2.1 The Interface Header

Each interface is introduced with exactly one interface header of the form

`INTERFACE` *interface-name* [`BRAND` *brand*] [`IMPORTS` *list-of-imported-interfaces* `END`] `;`

The *interface-name* is used by various language-specific productions to create name spaces in which the types, exceptions, and constants defined in the interface are declared.

The optional *brand* is a quoted string of printable US-ASCII characters (using the codes from 0x20 to 0x7E, inclusive). It is included into the type UID hash for types and exceptions defined in the interface, and can be used to make the type UIDs for otherwise-identical interfaces distinct.

The optional *list-of-imported-interfaces* is a comma-separated list of fields, each of the form

interface-name [`FROM` *interface-file*]

where *interface-file* is the name of the file containing the interface definition. Importing an interface allows the current interface to mention the types, exceptions, and constants defined in the imported interface, by referring to them as *interface-name* `.` *type-or-value-name*

The graph of imported interfaces must be acyclic; that is, interfaces may not mutually refer to each other, either directly or through some other interfaces. If the optional `"`FROM *interface-file*`"` is not specified for an imported interface, a sensible site-dependent search policy is followed in an attempt to locate that interface, typically looking down a path (environment variable `ILUPATH` on POSIX systems) of directories for a file with the name '*interface-name*`.isl`'.

2.2.2 Interface Directives

Interface directives are experimental and might be changed in the next release.

After the header and before the types, an interface can have an arbitrary list of interface directives of the form

`DIRECTIVE-EXPERIMENTAL` *list-of-directives* `;`

where *list-of-directives* is a comma separated list of quoted strings.

In certain stubbers the *list-of-directives* of some directive statements may contain language specific instructions about the mapping. Tools which convey special meaning to directive statements currently ignore directive statements which are not recognized by their first quoted string. It is typical that each directive statement is recognized only by tools in a few of the ILU languages; Directive statements influence mapping within one address space but unless understood by all languages don't change the inter address space interfaces.

2.2.3 Type Declarations

In general, a type is defined with a statement of the form

`TYPE` *type-name* = *type-reference* | *construction* [`TYPEID` *type-id-string*] `;`

The form `TYPE` *type-name* = *type-reference* is used when you want to rename an existing type to make its usage clear or give it a name in the current interface. A *type-reference* is just a *type-name*, or a reference to a type name defined in another interface: *interface-name.type-name*. The new name is then a 'nickname' for the previously defined type.

The optional `TYPEID` attribute is provided for use with CORBA interfaces, which allow specification of type ID information. Type IDs should conform to the syntax for URIs. It's probably safest **not** to use `TYPEID`, as ILU will automatically generate unique type ID's for your types.

2.2.3.1 Primitive types

The following type ''names'' are pre-defined:

- `INTEGER,` a 32-bit signed integer value;

- `SHORT INTEGER`, a 16-bit signed integer value;
- `LONG INTEGER`, a 64-bit signed integer value;
- `CARDINAL`, a 32-bit unsigned integer value;
- `SHORT CARDINAL`, a 16-bit unsigned integer value;
- `LONG CARDINAL`, a 64-bit unsigned integer value;
- `BYTE`, an unsigned 8-bit byte value;
- `BOOLEAN`, a logical value either True or False;
- `REAL`, an IEEE 64-bit double-precision floating-point value;
- `SHORT REAL`, an IEEE 32-bit single-precision floating-point value;
- `LONG REAL`, a 128-bit quadruple-precision floating-point value;
- `CHARACTER`, a 16-bit UNICODE/IS-10646 character; and
- `SHORT CHARACTER`, an 8-bit ISO 8859-1 character code (but excluding the octet 8_000).
- `PICKLE`, an opaque value containing some other value (see below).

There is also a special type `NULL`, which cannot be used directly; it has a single value, NULL.

`PICKLE` is an abstract type, values of which contain a 'pickled' or 'frozen' value of any other ISL type, and is thus used when a dynamically typed element is needed in an interface. Functionally, it is quite similar to the CORBA `any` type, but has more efficient semantics. `PICKLE` is only available if ILU has been configured with VARIANT support.

2.2.3.2 Constructor overview

The form `TYPE` *type-name = construction* is used when a user needs to define a new type. Several simple constructors for more complex data types are specified:

- `ARRAY`, a fixed-length N-dimensional array of some specified type;
- `SEQUENCE`, a variable-length one-dimensional array of some specified type;
- `RECORD`, a sequence of typed fields, each of which may be of a different type;
- `UNION`, one of a set of specified types;
- `OPTIONAL`, a union with `NULL`;
- `ENUMERATION`, a type consisting of an explicitly enumerated set of values;
- `OBJECT`, an ILU object type.
- `FIXEDPOINT`, a rational type with a fixed denominator value (*not yet implemented*);

In addition, the automatically-imported interface ILU defines the short sequence `CString` of short character.

2.2.3.3 Array Declarations

An `ARRAY` is a fixed-length N-dimensional array of some type. It is defined with a declaration of the form

`TYPE` *type-name* = `ARRAY OF` *dimension-list base-type-reference* `;`

where *dimension-list* is a comma-separated list of non-negative integers, each integer specifying the size of a dimension of the array, and *base-type-reference* is a *type-reference* to some other ILU type. For example,

```
TYPE SymbolTable = ARRAY OF 400 Symbol;
TYPE Matrix3030 = ARRAY OF 30, 30 REAL;
```

The total number of elements in the array may not exceed 4294967295 (2^32-1).

2.2.3.4 Sequence Declarations

A sequence is a variable-length one-dimensional array of some type. It is defined with a declaration of the form

`TYPE` *type-name* = [`SHORT`] `SEQUENCE OF` *base-type-reference* [`LIMIT` *size*] `;`

where *base-type-reference* is a *type-reference* to some other ILU type. If the `LIMIT` parameter *size* is used, it limits the sequences to having at most *size* elements; otherwise the sequences are limited to having at most 4294967295 (2^32-1) elements. Use of the `SHORT` modifier is shorthand for a `LIMIT` of 65535 (2^16-1). Use of the `LONG` modifier is not defined for sequences.

2.2.3.5 Generalized Array Declarations

This is a proposed language change, not yet accepted. *It is not supported in any of the language bindings.*

The existing language has a weakness: it cannot express coordinated multidimensional variable-length arrays. Coordinated means that there is only one length per dimension, regardless of how many arrays there are at that level. An example is a bitmap of variable height and width: all rows are the same length, and all columns are the same length.

A generalized array type is defined with a declaration of the form

`TYPE` *type-name* = `ARRAY` *dim* , ... *dim* `OF` *base-type-reference* `;`

where each *dim* is of the form

length | [`LIMIT` *maxlen* | `SHORT`]

A dimension can be given a fixed length by simply specifying that length. A variable-length dimension is either left blank (meaning the maximum length is 2**32-1), specified as `SHORT` (meaning the maximum length is 2**16-1), or given an explicit maximum length.

Note that putting the dimensions after the `OF` would create a syntactic ambiguity in some cases, concerning grouping of a `SHORT`.

2.2.3.6 Record Declarations

`TYPE` *type-name* = `RECORD` *fields...* `END ;`

where *fields* is a comma-separated list of *field*, which has the form

field-name : *field-type-reference*

A sample record declaration:

```
TYPE Symbol = RECORD
  name : string,
  ltype : TypeInfo,
  address : cardinal
END;
```

2.2.3.7 Union Declarations

A union is a type which may take on values of several different types. To be compliant with the CORBA notion of unions, the union declaration is much more baroque and complicated than it really should be. The declaration has the form:

`TYPE` *type-name* = [*tag-type*] `UNION` *arm-list* `END` [`OTHERS`] `;`

where *arm-list* is a comma-separated list of *arm*, each of the form:

[*union-case-name* :] *type-name* [*arm-valuator*]

where each *arm-valuator* is either of the form

= `DEFAULT`

or of the form

= *value-list* `END`

and where a *value-list* is a comma-separated list of constant values of the tag type. The tag type must be one of: `SHORT INTEGER`, `SHORT CARDINAL`, `INTEGER`, `CARDINAL`, `BYTE`, `BOOLEAN`, or an enumerated type. (We should also allow `SHORT CHARACTER` and `CHARACTER`.) The tag type is `SHORT INTEGER` if not explicitly specified.

A *arm-valuator* must be given for either all or none of the *arms*; if none, the *arms* are assigned single integral values, starting with 0. *arm-valuators* must be given if the tag type isn't numeric. All the values appearing in the *value-lists* of a union must be different from one another. `DEFAULT` can appear in at most one arm of a union type construction. `DEFAULT` and `OTHERS` cannot both appear in the same union.

A union value consists of a tag value, possibly paired with a second value. When the tag value is one that appears in, or is implicitly assigned to, an arm of the union type construction, the second value is of the type named in that arm. Otherwise, the union value is well-formed only if `DEFAULT` or `OTHERS` appears in the union type construction. If an arm is valued with `DEFAULT`, the second value is of that arm's type. If `OTHERS` appears, there is no second value; it is as if there were a default arm of some trivial type (like C's `void` or ML's `unit`).

A simple example:

```
TYPE StringOrInt = UNION ilu.CString, CARDINAL END;
```

A more complex example, that uses an explicit tag type, union case names, and a default arm:

```
TYPE ColorType = ENUMERATION RGB, CMY, HSV, YIQ, HLS END;
TYPE U2 = ColorType
  UNION
    rgb-field : RGBObject = RGB END,
    others : COLORObject = DEFAULT
  END;
```

The union case name is not guaranteed to be present in language-specific mappings.

ISL unions are logically (and sometimes actually, depending on the programming language) tagged. There is a difference between

```
TYPE T1 = UNION Bar, Baz END;
TYPE T2 = UNION Foo, T1 END;
```

and

```
TYPE T1 = UNION Bar, Baz END;
TYPE T2 = UNION Foo, Bar, Baz END;
```

2.2.3.8 Optional Declarations

A variable of type `OPTIONAL Foo` can have either a value of `Foo` or of type `NULL`. It is declared with the form

`TYPE` *type-name* = `OPTIONAL` *base-type-reference* `;`

This should be thought of as roughly equivalent to the declaration

`TYPE` *type-name* = `BOOLEAN UNION` *base-type-reference* = `TRUE END END OTHERS ;`

The difference is that `OPTIONAL` types are logically un-tagged. An optional value is not a pair of (`BOOLEAN`, *base-type-reference*); rather it is a single value, either a special, distinguished, "null" value or a value of the *base-type-reference*. There is thus no difference between

```
TYPE Bar = OPTIONAL Foo;
TYPE Baz = OPTIONAL Bar;
```

and

```
TYPE Bar = OPTIONAL Foo;
TYPE Baz = OPTIONAL Foo;
```

This type is not yet implemented.

2.2.3.9 Enumeration Declarations

An enumeration is an abstract type whose values are explicitly enumerated. It is declared with the form

`TYPE` *type-name* = `ENUMERATION` *values...* `END ;`

where *values* is a comma-separated list of value names, with optional value ID's that are constants of type `SHORT CARDINAL` that specify the value used to represent the enumeration value "on the wire".[3] *Use of value ID's is deprecated.*

value-name [= *value-id*]

For example,

```
TYPE TapeAction = ENUMERATION
  SkipRecord = 1,
  Rewind = 23,
  Backspace = 49,
  WriteEOF = 0
END;
```

All *value-names* and *value-IDs* must be unique within an enumeration. If *value-IDs* are not assigned explicitly, appropriate values will be assigned automatically in some unspecified way. An enumeration may have at most 65535 (2^16-1) values.

2.2.3.10 Object Type Declarations

Object types are described in the following way:

`TYPE` *type-name* = `OBJECT`

[3] Same integer in all protocols? Yep -- for now.

[`SINGLETON` *protocol-description-string*]
[`DOCUMENTATION` *documentation-string*]
[`COLLECTIBLE`]
[`OPTIONAL`]
[`TYPEID` *type-id-string*]
[`SUPERTYPES` *supertype-list* `END`]
[`METHODS` *method-list...* `END`]
[`BRAND` *brand-string*] `;`

The keyword `CLASS` is a deprecated synonym for `OBJECT`, and `SUPERCLASSES` is a deprecated synonym for `SUPERTYPES`. Also,

[`SUPERCLASS` *supertype-name*]

is a deprecated equivalent to

[`SUPERTYPES` *supertype-name* `END`]

The `SINGLETON` keyword specifies that instances of this type are singleton servers, and implies that the discriminator object (the subject of the call) should not be implicitly marshalled as the first argument in an RPC. This is typically used in describing an instance of an existing RPC service, which is to be modelled in ILU. The argument to `SINGLETON` is a string in the form of ILU ‘‘protocol-info’’, which specifies particular protocol-specific parameters to be used in implementing this object type ‘on the wire’. For example, the Sun RPC calendar manager would use a *protocol-description-string* of `"sunrpc_2_100068_3"`, indicating that it uses a Sun RPC program number of 100068 and a Sun RPC version of 3.

The optional *documentation-string* is a quoted string, which is passed on to language-specific bindings where possible, such as with the doc-string capability in Common Lisp.

The `COLLECTIBLE` keyword specifies that instances of this type are meant to be garbage collectible, and that methods necessary for this should be automatically added to its method suite. For an object type to be collectible, all ancestor object types must also be collectible.

The `OPTIONAL` keyword specifies that the language-specific `nil` value may be passed, instead of an instance of this object type, anywhere this object type is used. *This is a CORBA mis-feature, and its use is strongly deprecated. Better to explicitly use a different type constructed with the ILU* `OPTIONAL` *keyword.*

The optional *supertype-list* defines an inheritance relationship between the object types named in the list and the type *type-name*.

The optional *type-id-string* can be used to explicitly assign an MSTID for an object type. Doing so effectively *seal*s the object type; that is, changes to the structure of the object type will not be reflected in the MSTID, so version mismatches will not be caught automatically by ILU. This is a dangerous feature (mandated by CORBA).

The *brand-string* in the `BRAND` clause, if any, contributes an arbitrary tag to the structure of the type; omitting the `BRAND` clause is equivalent to giving one with the empty string. The brand is included in the type UID hash of the type, and thus gives the programmer a way to make two types distinct despite their otherwise having the same structure. The *brand-string* should be a quoted string of printable US-ASCII characters.

The *method-list* is a comma-separated list of procedure descriptions. All the methods of an object type have distinct names. This means that independently-developed supertypes might not be usable together.

Methods have the syntax:

[`FUNCTIONAL`] [`ASYNCHRONOUS`] *method-name* `(` [*args...*] `)`
  [`:` *return-type-reference*]
  [`RAISES` *exceptions...* `END`]
  [= *procedure-id*]
  [*documentation-string*]

where the discriminator (the implicit first argument to the method, the subject of the call, an instance of the object type in question) is not explicitly listed in the signature. Each method has zero or more arguments in a comma-separated list, each element of which is a colon-separated two-ple

[*argument-direction*] *argument-name* `:` [`SIBLING`] *argument-type-reference*

The `SIBLING` keyword may only appear on arguments of an object type, to indicate that the argument should be a sibling object to the discriminator of the method. The `FUNCTIONAL` keyword indicates that the method, for a given set of arguments, is idempotent (i.e., the side effects of one call are the same as the side effects of more than one call) and will always return the same result (or raise the same exception); this information may be used for caching of return values in the client side stubs. The optional *argument-direction* information is one of the three keywords `IN`, `OUT`, `INOUT`, specifying whether the parameter is being used as an input parameter, an output parameter, or both.

A method return type is allowed (again separated from the procedure argument list by a colon), and a list of possible exceptions may be specified as a comma-separated list of exception names, bracketed with the keywords RAISES and END.

The optional *procedure-id* field allows a service description to specify the procedure code that is used in the RPC request packet for this method. Procedure ID's are restricted to the range [0,65279], and must be unique within an interface. This may only be used in methods on objects marked with the `SINGLETON` attribute.

If a method is marked with the `ASYNCHRONOUS` keyword and does not return a value or raise an exception, the RPC method call of a surrogate instance will return after sending the request packet to the RPC partner, as the success of the call does not depend on the completion of the associated code. Other RPC methods will block in such a way as to allow the scheduler to handle other events while it is waiting for the call to complete, if the user has registered the appropriate scheduler hooks with the ILU runtime.

The optional *documentation-string* is a quoted string, which is passed on to language bindings for which it is meaningful, such as the doc-string capability in Common Lisp.

For example:

```
TYPE FancyString = OBJECT
  METHODS
    FUNCTIONAL Length () : cardinal,
    Substring (start : cardinal, end : cardinal) : string
      RAISES StartGreaterThanEnd, StartTooLarge, EndTooLarge END,
    Char (index : cardinal) : character
      RAISES BadIndex END
  END;
```

Note that the object language in ILU is not intended to be used to fully *define* an object type, but rather to *describe* it in a simple language that can be transformed into the different object type definition systems of several other languages.

2.2.3.11 Fixed-point Declarations

A `FIXEDPOINT` type is a rational type with a fixed denominator. It is defined with a declaration of the form

> `TYPE` *type-name* `= FIXEDPOINT [ MIN-NUMERATOR` *min-num-value* `] [ MAX-NUMERATOR` *max-num-value* `] [` DENOMINATOR *denom-value* `] ;`

where *denom-value* specifies the denominator of all values of the type, the optional *min-num-value* specifies the minimum numerator value allowed for values of the type, and the optional *max-num-value* specifies the maximum numerator value allowed for values of the type. If *denom-value* is negative, it is interpreted as the reciprocal of its absolute value. If either *min-num-value* or *max-num-value* is not specified, the numerator value is not bounded in that direction. If the denominator is not specified, it defaults to 1.

This type is still experimental and may not be supported in any of the ILU language bindings.

2.2.4 Exception Declarations

Exceptions in ILU are raised by ILU methods. They allow error conditions to be signalled back to the calling code. They are declared with a statement of the form:

> `EXCEPTION` *exception-name* [`:` *type-reference*] [*documentation-string*] `;`

The optional *type-reference* part of the declaration allows the exception to have an associated value, to be used in interpretation of the exception. For example, an exception BadFilename might have the type `ilu.CString`, so that the actual bad filename can be associated with the exception:

The optional *documentation-string* is a quoted string, which is passed on to language bindings for which it is meaningful, such as the doc-string capability in Common Lisp.

```
TYPE Filename = ilu.CString;
```

```
EXCEPTION BadFileName : Filename "The value is the bad filename";
```

Because of the uncertain nature of life in distributed systems, the pre-defined exception `ilu.ProtocolError` (defined in the ILU interface) may be raised by any method, to indicate that the method could not be handled, for some reason. It has the following form:

```
TYPE ProtocolErrorDetail = ENUMERATION
                NoSuchClassAtServer = 1,
                BrandMismatch = 2,
                NoSuchMethodOnClass = 3,
                InvalidArguments = 4,
                UnknownObjectInstance = 5,
                UnreachableModule = 6,
                RequestRejectedByModule = 7,
                TimeoutOnRequest = 8,
                UnknownError = 9
        END;

EXCEPTION ProtocolError : ProtocolErrorDetail;
```

Signalling of `ProtocolError` is never done by user-written server code; it is reserved to the transport and runtime layers of ILU.

2.2.5 Constant Declarations

For convenience of interface design, constant values for certain simple types may be defined in ISL with statements of the form

`CONSTANT` *constant-name* `:` *constant-type* = *constant-value* `;`

2.2.5.1 Integer, Cardinal, and Byte Constants

A *constant-value* for types that are sub-types of `INTEGER`, `CARDINAL`, or `BYTE` is specified with the syntax

[*sign*] [*base-indicator*] *digits*

where the optional *base-indicator* allows selection of bases 2, 8, 10 or 16. It is a digit '0' (zero) followed by either the character 'B' for base 2, 'X' for base 16, 'O' (oh) for base 8, or 'D' for base 10. The *sign* is only valid for subtypes of `INTEGER`; it is either '+' or '-'; if not specified, '+' is assumed. The *base-indicator* and *digits* fields are case-insensitive.

2.2.5.2 Real Constants

A *constant-value* for subtypes of `REAL` has the syntax:

[*sign*] *integer* . *fraction* [`e` *exponent*] where *integer* and *fraction* are sequences of decimal digits, *sign* is either '+' or '-' ('+' is the default), and *exponent* is the power of 10 which the rest of the value is multiplied by (defaults to 0).

2.2.5.3 ilu.CString Constants

A *constant-value* for a sub-type of `ilu.CString` has the form

`"`*characters*`"`

where *characters* are any ISO-Latin-1 characters except for 8_000. The escape character is defined to be '#' (hash). The escape character may occur in the string only in the following ways:

#" -- a single double-quote character
-- a single escape character
#*hex-digithex-digit* -- the octet 16_*hex-digithex-digit*
#n -- newline
#r -- carriage return

2.2.5.4 Examples of Constants

```
CONSTANT Newline : byte = 10;
CONSTANT Pi : short real = 3.14159;
CONSTANT Big : long real = -1.1349e27;  (*  -1.1349 * 10**27   *)
TYPE Filename = ilu.CString;
CONSTANT MyLogin : Filename = "~/.login";
CONSTANT Prompt : ilu.CString = "OK#n ";
CONSTANT HeapBound : cardinal = 0xFFFF39a0;
CONSTANT Pattern1 : cardinal = 0b000001000001;
```

2.3 ilu.isl

The standard interface `ilu` can be found in the file '*ILUHOME*`/interfaces/ilu.isl`'; it is maintained as '*ILUHOME*`/src/stubbers/parser/ilu.isl`'. Here are its contents:

```
INTERFACE ilu BRAND "version 2";

(*
Copyright (c) 1991-1997 Xerox Corporation.  All Rights Reserved.

Unlimited use, reproduction, and distribution of this software is
permitted.  Any copy of this software must include both the above
copyright notice of Xerox Corporation and this paragraph.  Any
distribution of this software must comply with all applicable United
States export control laws.  This software is made available AS IS,
and XEROX CORPORATION DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED,
INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE, AND NOTWITHSTANDING ANY OTHER
PROVISION CONTAINED HEREIN, ANY LIABILITY FOR DAMAGES RESULTING FROM
```

```
THE SOFTWARE OR ITS USE IS EXPRESSLY DISCLAIMED, WHETHER ARISING IN
CONTRACT, TORT (INCLUDING NEGLIGENCE) OR STRICT LIABILITY, EVEN IF
XEROX CORPORATION IS ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

$Id: isl.tim,v 1.53 1999/01/20 04:08:56 janssen Exp $
*)

TYPE CString = SEQUENCE OF SHORT CHARACTER;

TYPE CORBA-Object = OBJECT OPTIONAL TYPEID "IDL:omg.com/CORBA/Object:1.0";
```

2.4 Using OMG IDL with ILU

ILU allows the use of CORBA OMG IDL[4] instead of ILU ISL. It does this by translating the OMG IDL to its equivalent form in ISL. Most ILU tools will do this automatically, based on whether the suffix of the filename argument is '`.isl`' or '`.idl`'. In addition, the program `idl2isl` can be invoked explicitly to create an ISL version of an OMG IDL interface file. The program idl2isl translates from IDL to ISL. IDL is the Interface Definition Language defined by the Object Management Group.[5]

2.4.1 Translation

On the whole, the translation from IDL to ISL is a straightforward change of syntax. There are a few cases, however, where a bit more is needed.

2.4.1.1 Anonymous types

OMG IDL allows type declarators to be used in certain places in the syntax (for example, struct members and operation parameters). ISL does not; it requires a type name in the corresponding situations. As a result, it is sometimes necessary for the translator to introduce a name in the ISL output for those types that are anonymous in the OMG IDL input. These names are always of the form `AnonType-`*nnn*`-`, where *nnn* is an integer.

For example, the OMG IDL declaration

```
struct str {
long f1;
```

[4] OMG IDL is defined in: The Common Object Request Broker: Architecture and Specification, OMG Document Number 91.12.1, Revision 1.1

[5] The program's front end is derived from the Interface Definition Language Compiler Front End from SunSoft, Inc. See the file '`src/stubbers/idl2isl/Sun-parser/docs/COPYRIGHT`' in the ILU distribution.

```
long f2[5];
};
```

is translated into the following ISL:

```
TYPE AnonType-1- = ARRAY OF 5 INTEGER;
TYPE str = RECORD
f1 : INTEGER,
f2 : AnonType-1-
END;
```

2.4.1.2 Topmodules mode

When the translator is in this mode (which it is by default), only `module` declarations are allowed at the topmost level. Each `module` translates into an `INTERFACE` declaration in ISL, and the declarations inside each `module` go into the corresponding ISL `INTERFACE`.

If the translator is not in this mode, all the declarations in the IDL file go into one ISL `INTERFACE` whose name is taken from the OMG IDL input filename, less the '`.idl`' suffix.

2.4.1.3 Imports mode

When the translator is in this mode (which it is by default), `#include` preprocessor directives are, roughly speaking, turned into ISL `IMPORT` statements. This mode allows for separate compilation (stub generation) of interfaces. There are some restrictions: the `#include` directives must occur before any declarations in the file, and the files that are included must not be fragments. That is, each must consist of a sequence of whole declarations (more specifically, `module` declarations if in `topmodules` mode). The included files may in turn include other files.

If the translator is not in this mode, the input is considered to be the result of preprocessing the file first and textually substituting the included files, following the usual behavior of C and C++ compilers.

2.4.1.4 Unsupported constructs

If VARIANT support has not been configured in, the IDL type `any` is disallowed by the translator.

Use of `context` clauses on operations is not supported.

2.4.2 Manual Invocation of `idl2isl`

The program is run automatically as an intermediate step by any of the ILU tools that take ISL files (normally ending in '`.isl`') if the filename ends in '`.idl`'.

The program may also be run directly, with the following arguments:

`idl2isl` { `-Wb,`*toggle* | `-Wb,!`*toggle* }* *source*`.idl`

In this case, it writes the ISL to its standard output. A toggle is set with an argument `-Wb,`*toggle* and cleared with an argument `-Wb,!`*toggle*. Toggle settings may also be effected by setting the environment variable '`IDL2ISL_OPTS`' to a comma-separated list of toggle names, each of which is either preceded by a '`!`' character (which clears it) or not (which sets it). Command-line arguments take precedence over the environment variable settings.

The toggles are:

- `dump` (default off): produce a dump of the abstract syntax tree. Used for debugging the translator itself.
- `imports` (default on): set the `imports` mode on (explained below).
- `topmodules` (default on): set the `topmodules` mode on (explained below).

2.5 ISL Grammar

In this grammar, parentheses are used for grouping, vertical-bar indicates selection, braces indicated optionality, quotation marks indicate literal keywords or literal punctuation.

No whitespace is allowed between the parts of a `radix`, `number`, or `quoted-string`. Aside from that, whitespace is used to separate fields where necessary, and excess whitespace is ignored outside of `quoted-strings`.

Three primitives are used:

- *name-string*, which is a string consisting of decimal digits, upper and lower-case letters, and hyphens, beginning with a letter. It may not be a keyword, unless it is quoted with double-quotes.
- *string*, which is any sequence of characters.
- *digits*, which is a sequence of digits drawn from the digits for the particular radix. The default radix is decimal.

```
interface = interface-def | interface interface-def

interface-def = interface-declaration interface-directive-list other-declarations

interface-declaration = "INTERFACE" name-string
                        [ "BRAND" brand-string ]
                        [ "IMPORTS" import-list "END" ]
                        ";"

import-name = name-string [ "FROM" filename ]

import-list = import-name | import-list "," import-name

interface-directive-list = [ interface-directive interface-directive-list
]

interface-directive = "DIRECTIVE-EXPERIMENTAL" quoted-string-list ";"
```

```
quoted-string-list = quoted-string | quoted-string-list "," quoted-string

other-declarations = [ other-declaration other-declarations ]

other-declaration = constant-decl | exception-decl | type-decl

constant-decl = "CONSTANT" name-string ":" ( integer-const
                                           | cardinal-const
                                           | boolean-const
                                           | byte-const
                                           | float-const
                                           | string-const ) ";"

integer-const = [ "SHORT" | "LONG" ] "INTEGER" "=" [ sign ] number

boolean-const = "BOOLEAN" "=" boolean-value

cardinal-const = [ "SHORT" | "LONG" ] "CARDINAL" "=" number

byte-const = "BYTE" "=" number

float-const = [ "SHORT" | "LONG" ] "REAL" "="
              [sign] digits [ "." digits ] [ "e" digits ]

number = [ radix ] digits

radix = "0" ( binary | octal | hexadecimal )

binary = "b"

octal = "o"

hexadecimal = "x"

string-const = "ilu.CString" "=" quoted-string

exception-decl = "EXCEPTION" excp-name [ ":" type ] [ typeid ] [ doc-string
] ";"

excp-name = name-string

type-decl = "TYPE" name-string "=" ( type | type-cons ) [ typeid ]";"

type = primitive-type-name | [ name-string "." ] name-string

typeid = "TYPEID" type-id-string

primitive-type-name = "BYTE"
                    | [ "SHORT" | "LONG" ] "CARDINAL"
                    | [ "SHORT" | "LONG" ] "INTEGER"
```

```
                     | [ "SHORT" | "LONG" ] "REAL"
                     | [ "SHORT" ] "CHARACTER"
                     | "BOOLEAN"
                     | "PICKLE"

type-cons =   record-cons
                 | array-cons
                 | sequence-cons
                 | union-cons
                 | optional-cons
                 | enum-cons
                 | object-cons

record-cons = "RECORD" field-list "END"

field-list = field | field-list "," field

field = name-string ":" type

sequence-cons = [ "SHORT" ] "SEQUENCE" "OF" type [ "LIMIT" number ]

array-cons = "ARRAY" "OF" dimensions-list type

dimensions-list = number | dimensions-list "," number

union-cons = [ int-enum-or-boolean-type ] "UNION" union-field-list "END"

union-field-list = union-field | union-field-list "," union-field

union-field = [ field-name-string ":" ] type [ "=" const-list "END" ]

const-list = const | const-list "," const

const = number | enum-field-name | boolean-value

enum-cons = "ENUMERATION" enum-field-list

enum-field-list = enum-field | enum-field-list "," enum-field

enum-field = string

boolean-value = "TRUE" | "FALSE"

optional-cons = "OPTIONAL" type

object-cons = "OBJECT" object-attributes

object-attributes = object-feature | object-attributes object-feature

object-feature =  "SINGLETON" singleton-protocol-info
```

```
                 | "COLLECTIBLE"
                 | "OPTIONAL"
                 | "DOCUMENTATION" doc-string
                 | "BRAND" brand-string
                 | "SUPERTYPES" supertype-list "END"
                 | "METHODS" method-list "END"

supertype-list = type | supertype-list "," type

singleton-protocol-info = quoted-string

method-list = method | method-list "," method

method = [ "FUNCTIONAL" | "ASYNCHRONOUS" ] name-string
         arguments [ ":" return-type ] [ "RAISES" exception-list "END"]
         [ doc-string ]

return-type = type

exception-list = excp-name | exception-list "," excp-name

arguments = "(" [ argument-list ] ")"

argument-list = argument | argument-list "," argument

argument = [ "IN" | "OUT" | "INOUT" ] name-string ":" [ "SIBLING" ] type

brand-string = printable-ascii-quoted-string

doc-string = quoted-string

quoted-string = "\"" string "\""

printable-ascii-quoted-string = "\"" string consisting of only printable ASCII charac-
ters "\""

type-id-string = "\"" scheme-name ":" string "\""

scheme-name = string-without-colon-char
```

3 Using ILU with Standard C

3.1 Introduction

This document is for the C programmer who wishes to use ILU. (By C, we mean the language defined in the ISO/ANSI standard, not 'K&R C', or 'Portable C'.) The following sections will show how ILU is mapped into C constructs and how both C clients and servers are generated and built.

Using ILU with C is intended to be compatible with the OMG CORBA specification. That is, all of the naming and stub generation comply with the Common Object Request Broker Architecture, revision 2.0.[1]

Note that ILU does not support non-ANSI variants of the C language. In particular, it relies on having prototypes, all C library functions, and the capabilities of the C pre-processor.

When functions are described in this section, they are sometimes accompanied by *locking comments*, which describe the locking invariants maintained by ILU on a threaded system. See the file '*ILUHOME*`/include/iluxport.h`' for more information on this locking scheme, and the types of locking comments used.

A number of macros are used in function descriptions, to indicated optional arguments, and ownership of potentially malloc'ed objects. The macro `OPTIONAL(`*type*`)` means that the value is either of the type indicated by *type*, or the value `NULL`. This macro may only be used with pointer types. The macro `RETAIN(`*type*`)` indicates, when used on a parameter, that the caller retains ownership of the value, and when used in the result position, that the called function retains ownership of the value. The macro `PASS(`*type*`)` indicates, when used on a parameter, that the caller is passing ownership of the storage to the called function, and when used in the result position, that the called function is passing ownership of the called value to the caller. The macro `GLOBAL(`*type*`)` means that neither the caller nor the calling function owns the storage.

---

[1] The Common Object Request Broker: Architecture and Specification, revision 2.0, Draft July 1995

3.2 The ISL Mapping to C

3.2.1 Names

In general, ILU constructs C names from ISL names by replacing hyphens with underscores. Type names and class names are prepended with their interface name. For example, for the ISL type `T-1` in interface `I`, the generated name of the C type would be `I_T_1`.

Method name prefixes are specified by CORBA to be *module-name_interface-name*. C function names for ISL methods are composed of the generated class name prepended to the method name. For example, if the interface name is `X` and the class type name is `Y` and the ISL method name is `Z` then the C callable method name will be `X_Y_Z`. ILU C servers for this method must implement a function called `server_X_Y_Z`.

For field names within records and unions, hyphens are replaced with underscores.

3.2.2 Interface

The ISL `interface` is mapped to a prefix for all generated type names, constant names, and exception names, by replacing all hyphens in the interface name with underscore characters.

3.2.3 Basic Types

The following basic ISL types have the corresponding mappings in C, as specified by the CORBA 2.0 standard mapping for C:

- ISL `BOOLEAN` maps to C `CORBA_boolean`
- ISL `BYTE` maps to C `CORBA_octet`
- ISL `CHARACTER` maps to C `CORBA_wchar`
- ISL `SHORT CHARACTER` maps to C `CORBA_char`
- ISL `CARDINAL` maps to C `CORBA_unsigned_long`
- ISL `SHORT CARDINAL` maps to C `CORBA_unsigned_short`
- ISL `LONG CARDINAL` maps to C `CORBA_unsigned_long_long`
- ISL `INTEGER` maps to C `CORBA_long`
- ISL `SHORT INTEGER` maps to C `CORBA_short`
- ISL `LONG INTEGER` maps to C `CORBA_long_long`
- ISL `REAL` maps to C `CORBA_double`
- ISL `SHORT REAL` maps to C `CORBA_float`
- ISL `LONG REAL` maps to C `CORBA_long_double`

3.2.4 Constants

ISL constants are translated to C `const` expressions initialized to the specified value. Constant names are prepended with their interface name, separated from the name of the constant with a hyphen character.

3.2.5 Strings and Characters

ISL `character` and `short character` types are represented with the ILU types `ilu_character`, which hold values of 16-bit Unicode, and `ilu_shortcharacter`, which hold values of 8-bit ISO Latin-1.

String sequences (`SEQUENCE OF SHORT CHARACTER` or `SEQUENCE OF CHARACTER`) are just arrays of the character codes for the characters, using either Latin-1 codes (for `SEQUENCE OF SHORT CHARACTER`), or ISO 10646 Unicode codes (for `SEQUENCE OF CHARACTER`). These sequences are terminated with a character code of zero. The terminating code is not counted in the length of the sequence.

3.2.6 Pickles and Typecodes

ILU pickles are mapped to opaque structures of the type `CORBA_any`, as per the CORBA specification for the type `any`. However, in ILU, the fields of the pickle are not directly accessible. Instead, the following utility functions are provided to manipulate pickles:

`PASS(CORBA_any *)` **ILU_C_Any_Create** (`RETAIN(CORBA_TypeCode)` *typecode*, `RETAIN(void *)` *value*, `RETAIN(CORBA_Environment *)` *env*) Function

Locking: n/a

Create a new pickle from a C *value* and *typecode*. The return value is heap-allocated.

`PASS(CORBA_any *)` **ILU_C_Any_Init** (`RETAIN(CORBA_any *)` *uninitialized-any*, `RETAIN(CORBA_TypeCode)` *typecode*, `RETAIN(void *)` *value*, `RETAIN(CORBA_Environment *)` *env*) Function

Locking: n/a

Given a pointer to an uninitialized `CORBA_any` value *uninitialized-any*, sets the typecode and value of the any to the specified *typecode* and *value*.

`PASS(CORBA_any *)` **ILU_C_Any_ResetValue** (`RETAIN(CORBA_any *)` *initialized-any*, `RETAIN(CORBA_TypeCode)` *typecode*, `RETAIN(void *)` *value*, `RETAIN(CORBA_Environment *)` *env*) Function

Locking: n/a

Given a pointer to a previously used `CORBA_any` value *initialized-any*, frees the current value and sets the typecode and value of the any to the specified *typecode* and *value*.

`PASS(void *)` **ILU_C_Any_Value** (`RETAIN(CORBA_any *)` *pickle*, `RETAIN(CORBA_Environment *)` *env*) Function

Locking: n/a

Returns the C value from the *pickle*. Returns NIL if the type of the value contained in the pickle is not 'known' to the ILU C runtime. Relatively expensive, as it involves several malloc's.

`CORBA_TypeCode` **ILU_C_Any_TypeCode** (`RETAIN(CORBA_any *)` *pickle*, `RETAIN(CORBA_Environment *)` *env*) Function

Locking: n/a

Retrieve the CORBA typecode of the value in the *pickle*. Returns NIL if the type of the value in the pickle is not registered with the ILU C runtime.

`PASS(CORBA_any *)` **ILU_C_Any_Duplicate** (`RETAIN(CORBA_any *)` *pickle*, `RETAIN(CORBA_Environment *)` *env*) Function

Locking: n/a

Make a copy of an existing pickle without 'looking inside'. This call will work even with pickle values that are of types not known to the ILU C runtime.

3.2.7 Constructed Types

3.2.7.1 Enumeration

ISL enumeration types are mapped C enum types, in an exception to the CORBA specification. Each element of the enumeration is named as <*interface*>_<*element-name*>. The C `enum` type is `typedef`'ed to the specified name for the type. For example, the ISL definition

```
INTERFACE Foo;
...
TYPE Color = ENUMERATION Red, Green, Blue END;
...
```

would produce the following C definition:

```
typedef enum { Foo_Red, Foo_Green, Foo_Blue } Foo_Color;
```

3.2.7.2 Array

Arrays are represented as C arrays.

3.2.7.3 Sequence

Sequence type names, as most type definitions, are formed with the interface name and the type name. Sequence instances are represented to the C programmer as a pointer to the sequence descriptor structure.

For each sequence type declared in the interface description, a pseudo-object sequence type is defined in C. These sequence types will hold any number of values of type sequence's *primary type*. For the sequence

```
INTERFACE I;
TYPE T2 = SEQUENCE OF T1;
```

the following functions are defined:

I_T2 * *I_T2*_**Create** (OPTIONAL(unsigned long) *length*, OPTIONAL(*T1* *) *initial-values*) — Sequence Method

This function creates and returns a pointer to a newly allocated instance of *T2*. If *length* is specified, but *initial-values* is not specified, enough space for *length* values of type *T1* is allocated in the sequence. If *initial-values* is specified, *length* is assumed to be the number of values pointed to by *initial-values*, and must be specified. Note that if type *T1* is a `character` or `short character` type, a pointer to a NIL-terminated sequence will be returned; otherwise, a normal CORBA sequence structure will be returned by reference.

`CORBA_unsigned_long` *I_T2*_**Length** (*I_T2* * *s*) — Sequence Method

Returns the length of *s*.

`void` *I_T2*_**Append** (*I_T2* * *s*, *T1 value*) — Sequence Method

Appends *value* to the end of *s*. This function will reallocate space and copy, if necessary.

`void` *I_T2*_**Push** (*I_T2* * *s*, *T1 value*) — Sequence Method

Pushes *value* on to the beginning of the sequence. This function will reallocate space and copy, if necessary.

`void` *I_T2*_**Pop** (*I_T2* * *s*, *T1* * *value-ptr*) — Sequence Method

Removes the first value from the sequence *s*, and places it in the location pointed to by *value-ptr*.

`void` *I_T2*_**Every** (*I_T2* * *s*, `void` (**func*)(*T1*, `void` *), `void` * *data*) — Sequence Method

Calls the function *func* on each element of *s* in sequence, passing *data* as the second argument to *func*.

I_T1 * *I_T2*_**Nth** (*I_T2* * *s*, `CORBA_unsigned_long` *n*) — Sequence Method

Returns the address of the *n*th element of the sequence *s*. Returns `ILU_NIL` if *n* is out of range.

`void` *I_T2*_**Init** (*I_T2* * *s*, OPTIONAL(`CORBA_unsigned_long`) *length*, OPTIONAL(*T1* *) *initial-values*) — Sequence Method

This function works like *T2*`_Create`, except that it takes a the address of an already-existing *T2* to initialize. This can be used to initialize instances of *T2* that have been stack-allocated.

`void` *I_T2*_**_Free** (*I_T2* `*` *s*) Sequence Method

Frees allocated storage used internally by *s*. Does not free *s* itself.

String sequences (`SEQUENCE OF SHORT CHARACTER` or `SEQUENCE OF CHARACTER`) are just arrays of the character codes for the characters, using either Latin-1 codes (for `SEQUENCE OF SHORT CHARACTER`), or ISO 10646 Unicode codes (for `SEQUENCE OF CHARACTER`). These sequences are terminated with a character code of zero. The terminating code is not counted in the length of the sequence. All other sequence types have a record structure, mandated by CORBA:

```
typedef struct I_T2 {
  unsigned long _maximum;
  unsigned long _length;
  I_T1 *_buffer;
} I_T2;
```

The field `_maximum` contains the number of elements pointed to by `_buffer`. The field `_length` indicates the number of valid or useful elements pointed to by `_buffer`.

For example, the ISL specification

```
INTERFACE I;

TYPE iseq = SEQUENCE OF INTEGER;
```

would have in its C mapping the type

```
typedef struct I_iseq {
  unsigned long _maximum;
  unsigned long _length;
  ilu_integer *_buffer;
} I_iseq;
```

In a client program, a pointer to this type would be instantiated and initialized by calling the type specific sequence creation function generated for the sequence, e.g.

```
        ...
    I_O h;
    ILU_C_ENVIRONMENT s;
    I_iseq sq;
        ...
    sq = I_iseq_Create (0, NULL);
    I_iseq_Append (&sq, 4);
        ...
```

3.2.7.4 Record

Records map directly into corresponding C structures.

3.2.7.5 Union

Because of the somewhat baroque CORBA concept of union types, unions may take one of several forms.

Generally, ILU unions in C consist of a struct with two members: the type discriminator (a member named ‘‘`_d`’’), and a union (a member named ‘‘`_u`’’) of the possible values. In a simple ISL union that does not name the elements, the union member names are derived from the ISL data types which compose the union. For example, if the ISL type in interface `I` is `TYPE u1 = UNION INTEGER, SHORT REAL END;` the generated C struct would be

```
struct _I_u1_union {
        CORBA_short _d;
        union {
                CORBA_long integer;     /* 0 */
                CORBA_float shortreal;  /* 1 */
        } _u;
};
typedef struct _I_u1_union I_u1;
```

Note the discriminator `_d` may take on the values of `0`, for the `integer` field, or `1`, for the `shortreal` field.

In more complex union forms, the user may specify the type of the discriminator as well as the member names and which member corresponds to which discriminator value. Consider the following ISL example:

```
INTERFACE I;
TYPE e1 = ENUMERATION red, blue, green, yellow, orange END;
TYPE u1 = e1 UNION
 a : INTEGER = red, green END,
 b : SHORT REAL = blue END,
 c : REAL = DEFAULT
END;
```

The generated union is:

```
typedef struct _I_u1_union I_u1;
typedef enum { I_red = 0, I_blue = 1, I_green = 2, I_yellow = 3, I_orange
= 4 } I_e1;
struct _I_u1_union {
  I_e1 _d;
  union {
    CORBA_long a;       /* I_red, I_green */
    CORBA_float b;      /* I_blue */
    CORBA_double c;     /* DEFAULT */
  } _u;
};
```

This example shows that the discriminator type is to be `I_e1` and that the member names are to be `a`, `b`, and `c`. When the discriminator has the value `I_red` or `I_green` the member `a` has a valid value and the type is interpreted to be `CORBA_long`. When the discriminator has the value `I_green` the member `b` has a valid value and the type is interpreted to be `CORBA_float`. If the discriminator has any other value, the member `c` is expected to have a valid value and the type is interpreted to be `CORBA_double`.

Discriminator types may be `INTEGER`, `CARDINAL`, `ENUMERATION`, `SHORT CARDINAL`, or `SHORT INTEGER`. The default for an unspecified discriminator is `SHORT INTEGER`.

3.2.7.6 Optional

An ISL `OPTIONAL` type maps either to the same C type as its base type, if that base type is represented with an C pointer type, or to a pointer to that base type, if it is not represented with a C pointer type.

3.2.8 Object Types

3.2.8.1 Surrogate and True Objects

ILU uses an object system embedded into C. The C type of objects in this system is `ILU_C_Object*`; the identifiers `ILU_C_OBJECT` and `CORBA_Object` can also be used for this type. We recommend the use of `ILU_C_OBJECT`, as it is a macro that expands to the identifier for the C object type specified by the version of CORBA being used (currently 2.0).

Since C has no subtyping relationship isomorphic to that of ISL object types, a more liberal approximation is used: all ISL object types map to the same C type, `CORBA_Object`. However, a `typedef` is emitted for each ISL object type, so an object-type-specific C type name may be used to express intent. For conformance with CORBA, the C name for ISL type *I.T* is *I_T*.

The C mapping of an object type includes a set of generic functions for the methods introduced at that object type. The name of the generic function for the method *M* introduced at object type *I.T* is *I_T_M*. This generic function is used to invoke method *M* on objects of type *I.T* (naturally including objects that also have types that are subtypes of *I.T*). The first argument to a generic function is an object instance (`ILU_C_Object *`) that should have type *I.T* (among possibly others).

An implementation of an ISL object type is commonly referred to as a *class*. Classes are represented in the ILU C runtime as the C type `ILU_C_Class`. To create an `ILU_C_Class`, the application calls an object-type-specific function that is part of the stubs. The class-creation procedure named *I_T*__`MakeClass`, which is declared in generated file '*I*`.h`', makes a class for objects that implement object type *I.T* (and thus all its supertypes). This procedure takes as arguments a set of C procedures that are the implementations of the methods of that class, plus a finalization procedure. The finalization procedure is given access to the private data of the object after the object is destroyed.

For each object type *I.T*, the generated server-side stub module for *I* creates a default true class, unless the true code has been generated with the `-nodefaulttrueclass`. Linking with this server-side stub requires the application to supply the procedures that implement the methods of this default class. Those procedures are named `server_`*I_T_M*, for each method *M* of *I.T*. A finalization procedure can be associated with the default class by invoking *I_T*`__SetDefaultClassFinalization`. Implementations of true objects typically just use this default class, since the methods for this class have to be provided in any case. The function *I_T*`__CreateTrue` will create an true instance using the default true class for the object type *I_T*. *I_T*`__CreateTrue` methods take an *instance handle*, a server on which to maintain the object, and arbitrary user data, and create and return the true instance of the object. An alternate version of *I_T*`__CreateTrue`, called *I_T*`__OTCreateTrue`, is provided for use inside the scope of an object table's incarnation procedure. Also, a generic creation procedure, `ILU_C_CreateTrueObject`, is declared in '`iluchdrs.h`' for application use. The default true class can be registered or changed with the function *I_T*`__SetDefaultClass`, which returns the previous setting of the default true class.

Surrogate instances generally use another automatically-constructed class, though custom surrogate classes may be registered with the procedure `ILU_C_RegisterCustomSurrogateType`. Surrogate instances are typically either received as reply values from calls or parameters to calls, or reified from one or more strings with a *binding procedure*. (A *binding procedure* is a procedure that takes some name for an object instance, and returns the actual instance.) There are a number of binding procedures available. The simple binding interface to C offers the function `ILU_C_LookupObject`, which takes an object ID and a type, and returns the registered object with that ID, if any. The function `CORBA_ORB_string_to_object` will take a URL for an instance, in any of the supported URL forms, and return an `ILU_C_OBJECT` instance. The function `ILU_C_SBHToObject` is similar to `CORBA_ORB_string_to_object`, except that an expected object type may also be specified, to constrain the process. In addition, the ILU-generated stubs will provide a function called *I_T*`__CreateFromSBH`, which will either find or create an instance of the specified type, with the specified parameters.

In general, for any object type *T*, the following C functions are defined:

ILU_C_FinalizationProc — C Procedure Type

Locking: caller may have any locks -- this means that this finalization procedure can do almost nothing ILU-related without potentially violating locking constraints

The finalization procedure for a class of objects. It takes a single parameter, the `void * "instance data"` pointer provided at the creation of the object being finalized. Ownership of (instanceData) is passed to this procedure. It has a return type of `void`.

`ILU_C_Class` ***T*__MakeClass** (*method-1-type method-1-proc*, ... *method-N-type method-N-proc*, `ILU_C_FinalizationProc _finalize`) — Function

Locking: Main Invariant holds

Creates a C class of objects that export object type *T* (and all its supertypes), given implementations for all the methods of that type. `ILU_C_CreateTrueObject` can then be called to create instances of this class.

`ILU_C_Class` *T*__**SetDefaultClass** (`ILU_C_Class c`) Function

Locking: Main Invariant holds

Sets the default true class of the type *T* to be *c*, and returns the previous default true class setting.

`void` *T*__**SetDefaultClassFinalization** (`ILU_C_FinalizationProc f`) Function

Locking: Main Invariant holds

Sets the finalization method of the default class for object type *T*.

`OPTIONAL(`*T*`)` *T*__**CreateTrue** (`OPTIONAL(RETAIN(char *))` *instance-id*, `OPTIONAL(GLOBAL(ilu_Server))` *server*, `OPTIONAL(PASS(void *))` *user-data*) Function

Locking: Main Invariant holds

Creates an instance of the default class for type *T*, exporting it with instance-id *instance-id*, exporting it via server *server*, associating the value *user-data* with it. If *instance-id* is not specified, a server-relative instance-id will be assigned automatically. If *server* is not specified, a default server will be created automatically.

`OPTIONAL(`*T*`)` *T*__**OTCreateTrue** (`RETAIN(char *)` *instance-id*, `GLOBAL(ilu_Server)` *server*, `OPTIONAL(PASS(void *))` *user-data*) Function

Locking: Inside(*server*, *T*)

Similar to *T*`__CreateTrue()`, but designed to be used within the `ot_object_of_ih` function of an object table (Section 3.3.1.2 [Servers and Ports], page 50). Requires kernel server locks to be held before invocation.

Creates an instance of the default class for type *T*, exporting it with instance-id *instance-id*, exporting it via server *server*, associating the value *user-data* with it.

`OPTIONAL(`*T*`)` *T*__**CreateFromSBH** (`RETAIN(char *)` *sbh*, `RETAIN(CORBA_Environment *)` *Env*) Function

Locking: Main Invariant holds

Finds or creates an instance of *T*, using the given object reference.

`extern ilu_Class` *T*__**MSType** Class Var

A value, of type `ilu_Class` (which, despite its poorly-chosen name, identifies an object type, not a class), that identifies the object type *T*.

In the following example, the ILU definition is:

```
INTERFACE I;

TYPE T = OBJECT
  METHODS
```

```
    M ( r : REAL ) : INTEGER
  END;
```

This definition defines an interface `I`, an object type `T`, and a method `M`. The method `M` takes a `REAL` as an argument and returns an `INTEGER` result. The generated C header file would include the following statements:

```
typedef ILU_C_OBJECT I_T;

ILU_C_Class I_T__MakeClass(
  ilu_integer (*I_T_M__Impl)
     (I_T _handle, ilu_real r, ILU_C_ENVIRONMENT *_status),
  ILU_C_FinalizationProc _finalize);

I_T I_T__CreateFromSBH (char *sbh, ILU_C_ENVIRONMENT *Env);
I_T I_T__CreateTrue (ilu_string ih, ilu_Server server, void *user_data);
I_T I_T__OTCreateTrue (ilu_string ih, ilu_Server server, void *user_data);

ilu_integer I_T_M (I_T, ilu_real, ILU_C_ENVIRONMENT *);

ilu_integer server_I_T_M (I_T, ilu_real, ILU_C_ENVIRONMENT *);
```

The functions `I_T__CreateFromSBH`, `I_T__CreateTrue`, and `I_T__OTCreateTrue` are used to create instances of the class `I_T`. `I_T__CreateFromSBH` is used by clients. `I_T__CreateTrue` is used by servers for normal circumstances, and `I_T__OTCreateTrue` is used in object table implementations; both return objects whose `M` method is implemented by `server_I_T_M`. Alternatively, servers and/or object tables could use `I_T__MakeClass` and `ILU_C_CreateTrueObject`. The pointer returned in each case is the object instance and must be passed with each method invocation.

3.2.8.2 Interface Inheritance

Through interface inheritance, an object type may participate in the behaviors of several different types that it inherits from. These types are called *ancestors* of the object type. In C, an object type supplies all methods either defined directly on that type, or on any of its ancestor types.

Consider the following example:

```
INTERFACE I2;

EXCEPTION E1;

TYPE T1 = OBJECT
  METHODS
    M1 (a : ilu.CString) : REAL RAISES E1 END
  END;

TYPE T2 = OBJECT
```

```
    METHODS
      M2 ( a : INTEGER, Out b : INTEGER )
    END;

  TYPE T3 = OBJECT SUPERTYPES T1, T2 END
    METHODS
      M3 ( a : INTEGER )
    END;
```

The object type `T3` inherits from the object types `T1` and `T2`. Thus, eight C procedures are relevant to the interface `I2`: the three generic functions `I2_T1_M1`, `I2_T2_M2`, and `I2_T3_M3`, and the five default method implementations `server_I2_T1_M1`, `server_I2_T2_M2`, `server_I2_T3_M1`, `server_I2_T3_M2`, and `server_I2_T3_M3`. A module that implements true instances of `T3` using the default class would define the last three procedures (the other two default method implementations, for messy reasons described in the next paragraph). A client uses only the three generic functions.

Sadly, the current state of the C-stubber causes an additional complexity for server implementors. '`I2-true.c`' contains the server-side stubs ("skeletons", in OMG parlance) needed in any program that implements any object type that is a subtype of any object type defined in '`I2.isl`'. '`I2-true.c`' also contains the code that creates the default classes for all the object types defined in '`I2.isl`'; this code makes external references to the default implementation procedures, thus requiring any program that links with '`I2-true.o`' to supply those default implementation procedures --- even if those default classes are not used. A simple workaround is to supply dummy procedures to satisfy the linker. The stubs can also be generated with the command-line option `-nodefaulttrueclass`, which will prevent generation of the code that creates the default true classes. However, if this technique is used, be aware that either a default true class must be registered manually, or true instances must be created with `ILU_C_CreateTrueObject`.

3.2.8.3 Accessing the String Binding Handle, IOR, or Object ID

Several functions are provided to give access to various identifiers of an instance. The function `ILU_C_SBHOfObject` will return the ILU URL for an instance; the function `ILU_C_IOROfObject` will return the CORBA URL for an object, if support for IIOP is configured into ILU; the function `CORBA_ORB_object_to_string` will return either the IOR, if `IIOP` support has been configured in, or the SBH, if not; the two parts of the object ID, the server id and the instance handle, may be obtained with a call on `ILU_C_IDOfObject`. See the API reference section of this chapter for more information on these functions.

3.2.8.4 Distinguishing Between Local and Remote Instances

It is occasionally useful to distinguish between local and remote instances. There are three cases here: the case where the instance is a true object, the case where the instance is a surrogate for a true instance implemented in another language in the same address space, and the case where the instance is a surrogate for a true instance in a different address space. There is currently no good way to distinguish these cases in the C runtime.

3.2.8.5 Object Implementation

This information is provided for those interested in the implementation of the C object system. It is **not** *guaranteed to remain the same from release to release.*

The interface '*ILUSRC*/runtime/c/ilucstub.h' contains the C declarations relevant to the material here.

A C class is represented by a pointer to a struct that holds a finalization procedure and a dispatch table. The dispatch table is an array of sections, one from each object type implemented by the class. Each section is a struct containing the `ilu_Class` (remember, this identifies a type, not a class) that this section is for, and an array of procedure pointers, one per method introduced at that object type.

For each method directly defined in the type, a generic function is defined in the common code for its interface, which dispatches to the appropriate method. It does this by walking down the dispatch table for the object's class, until it finds a section that contains the appropriate `ilu_Class` value (that is, the value of `ilu_Class` that matches the `ilu_Class` at which this method was introduced), then calling the method pointer which is indexed in the section's array of method pointers by the index of the method. The generic functions have the correct type signature for the method. They can be referenced with the `&` operator.

3.2.8.6 Methods

All ISL methods of an object type map to C functions which operate on instances of the C object system as described above. `IN`, `OUT`, and `INOUT` parameters appear in the C function in the same order as they appear in the ISL definition of the function.

`ASYNCHRONOUS` methods have no return values and raise no user-specified exceptions. They may return before the completion of the true method. `FUNCTIONAL` methods are not cached by the C ILU runtime.

In addition to its specified arguments, the methods `I_T_M` and `server_I_T_M` take an instance of the type `I_T` and a reference to a variable of type `ILU_C_ENVIRONMENT *`, which is a macro defined to be the appropriate CORBA environment type, and is used to return exception codes. The environment struct pointed to by the environment argument must be instantiated in a client; its address is passed as the last argument to each method. True procedures must expect a pointer to this structure as the last argument.

For instance, the C client calling the method for `M` might be as follows:

```
#include "I.h"

int main (int ac, char **av)
{
  double atof( );
  I_T inst;
  CORBA_long result;
  CORBA_double f;
  ILU_C_ENVIRONMENT ev;
```

```
    I__Initialize( );
    f = atof (av[1]);
    inst = I_T__CreateFromSBH (av[2], &ev);
    if (!ILU_C_SUCCESSFUL(&ev)) {
      printf( "CreateFromSBH raised exception <%s>\n",
        ILU_C_EXCEPTION_ID(&ev));
      return(1);
    }
    result = I_T_M (inst, f, &ev);
    if (!ILU_C_SUCCESSFUL(&ev)) {
      printf( "exception <%s> signalled on call to I_T_M\n",
        ILU_C_EXCEPTION_ID(&ev));
      return(2);
    }
    printf( "result is %d\n", result );
    return(0);
  }
```

Note the call on the interface-specific client initialization procdedure `I__Initialize`; these are described in a later section.

The string binding handle is obtained from standard input along with some floating-point value. The class specific function `I_T__CreateFromSBH` is then called to obtain the object instance. This function was passed the string binding handle, and a `CORBA_Environment` in which to report exceptions. The returned object instance is then passed as the first argument to the method `I_T_M`, along with the environment `ev`, and the single actual `CORBA_double` argument `f`. `I_T_M` returns an `CORBA_long` value which is placed in `result`.

The true implementation of the method `M` might use the default class, supplying the implementation of the one method as follows:

```
  ilu_integer server_I_T_M ( I_T h, ilu_real u, ILU_C_ENVIRONMENT *s )
  {
    return( (ilu_integer) (u + 1) );
  }
```

In this simple example, the corresponding server, or true, method computes some value to be returned. In this case it adds one to its `ilu_real` argument `u`, converts the value to an integer, and returns that value. Note that the server method, if not signalling any exceptions, may ignore the environment parameter.

3.2.8.7 Parameter Passing Considerations

Here is ILU's version of table 20 from the CORBA 2.0 spec. `T` is the C mapping of the type in question. The `Exn` column describes how exception parameters appear in the parameter-conveying member of a status struct.

```
  DataType        In      InOut   Out         Return    Exn
  --------        --      -----   ---         ------    ---
```

scalar	T	T*	T*	T	T*
optional	T	T*	T*	T	T*
object	T	T*	T*	T	T*
record, fixed	T*	T*	T*	T	T*
record, var	T*	T*	T**	T*	T*
union, fixed	T*	T*	T*	T	T*
union, var	T*	T*	T**	T*	T*
string	T	T*	T*	T	T*
other sequence	T*	T*	T**	T*	T*
array, fixed	T	T	T	T_slice*	T*
array, var	T	T	T_slice**	T_slice*	T*

3.2.8.8 Exceptions

C has no defined exception mechanism. As already indicated, exceptions are passed in ILU C by adding to the end of each method an additional status argument that can convey an exception code and a value of a type associated with that exception. To signal an exception, a method implementation sets the exception code and supplies the parameter value (if any).

An exception parameter is conveyed in the status argument as a C pointer; the parameter-conveying member is declared to be a `void *`. In particular, this pointer is a pointer to a value of the type that is the C mapping of the exception's ISL parameter. For an exception that has no parameter, the parameter-conveying member is not meaningful.

In the following example, the `div` method can raise the exception `DivideByZero`:

```
INTERFACE calc;

TYPE numerator = INTEGER;

EXCEPTION DivideByZero : numerator;

TYPE self = OBJECT
  METHODS
    Div( v1 : INTEGER, v2 : INTEGER ) : INTEGER RAISES DivideByZero END
  END;
```

The generated include file '`calc.h`' contains the exception definitions:

```
#ifndef __calc_h_
#define __calc_h_
/*
** this file was automatically generated for C
** from the interface spec calc.isl.
*/

#ifndef __ilu_c_h_
#include "ilu-c.h"
#endif
```

```
extern ILU_C_ExceptionCode    _calc__Exception_DivideByZero;
#define ex_calc_DivideByZero _calc__Exception_DivideByZero

typedef ilu_integer calc_numerator;
typedef calc_numerator calc_DivideByZero;

typedef ILU_C_OBJECT calc_self;

calc_self calc_self__CreateTrue ( char *id, ilu_Server server,
   void * user_data);
calc_self calc_self__CreateFromSBH ( char * sbh, ILU_C_ENVIRONMENT *Env );

ilu_integer calc_self_Div( calc_self, ilu_integer, ilu_integer,
   ILU_C_ENVIRONMENT *Env );

extern void calc__BindExceptionValue (ILU_C_ENVIRONMENT *, ilu_Exception,
...);

#endif
```

The method implementation for `Div` in the true module must detect the divide-by-zero condition and raise the exception:

```
long server_calc_self_Div (calc_self h, ilu_integer u, ilu_integer v,
                           ILU_C_ENVIRONMENT *s)
{
  calc_numerator n = 9;

  if ( v == 0 )
    {
      s->_major = ILU_C_USER_EXCEPTION;
      s->returnCode = ex_calc_DivideByZero;
      s->ptr = (void *) malloc(sizeof(calc_numerator));
      *((calc_numerator *) (s->ptr)) = n;
      s->freeRoutine = (void (*) (void *)) 0;
      return( u );
    }
  else
    return( u / v );
}
```

When freeing the parameter requires more than just freeing `s->ptr`, a non-NULL `s->freeRoutine` is provided that does the additional freeing; `s->freeRoutine` is given one argument, `s->ptr`, and returns `void`.

The generated stubs offer as a convenience a variadic procedure (`calc__BindExceptionValue`) that can be used to raise any exception declared in the interface. For an exception that has no parameter, this procedure takes just two actual arguments. For an exception with a parameter, the parameter value is given as the third actual argument, using the usual calling convention for passing `IN` arguments of its type. Using this procedure, the above code would be:

```
long server_calc_self_Div (calc_self h, ilu_integer u, ilu_integer v,
                           ILU_C_ENVIRONMENT *s)
{
  calc_numerator n = 9;

  if ( v == 0 )
    {
      calc__BindExceptionValue(s, ex_calc_DivideByZero, n);
      return( u );
    }
  else
    return( u / v );
}
```

The exception is sent back to the client, which can detect it thusly:

```
  ...
  calc_self instance;
  ILU_C_ENVIRONMENT s;
  ilu_integer i, j;
  ilu_integer val;
  ...
  instance = calc_self__CreateFromSBH (sbh, &s);

  if (! ILU_C_SUCCESSFUL(&s)) {
    fprintf (stderr, "CreateFromSBH(%s) raised %s\n",
      sbh, ILU_C_EXCEPTION_ID (&s) );
    exit(1);
  }

  val = calc_self_Div (instance, i, j, &s);

  /* check to see if an exception occured */

  if (! ILU_C_SUCCESSFUL(&s)) {
    /* report exception to user */
    char *p;

    p = ILU_C_EXCEPTION_ID (&s);

    if (p == ex_calc_DivideByZero) {
      calc_numerator *ip;
      ip = (calc_numerator *) ILU_C_EXCEPTION_VALUE (&s);
      fprintf (stderr, "%s signaled:  numerator = %d\n", p, *ip);
      }
    else {
      /* odd exception at this point */
```

```
            fprintf (stderr, "Unexpected <%s> on call to Div.\n", p);
            }
          /* free up any transient exception data */
          ILU_C_EXCEPTION_FREE (&s);
          }
        else {
          /* no exception - print the result */
          printf( "result is %d \n", val );
          }
        ...
```

For more complex exception types, it is often helpful to define a procedure in C specifically to raise them.

3.2.8.9 Garbage Collection of C Instances

Both true and surrogate instances of `ILU_C_OBJECT` are *reference-counted*; that is, each instance contains a counter which indicates how many uses are currently being made of that object. When you wish to use an object for another purpose, you should increment the reference count by calling the procedure `CORBA_Object_duplicate`, which will return a copy of the instance that you can use for the new purpose. When you are finished with an instance, you should call `CORBA_Object_release` to release your claim on the object's resources. When the reference count of the object returns to zero, the object is finalized.

For COLLECTIBLE true instances, the ILU kernel will maintain a distributed reference count on the instance automatically. That is, so long as any client of the server has a reference to the object, the true instance will not be finalized.

3.2.9 Issues in Mapping OMG IDL to C

The ILU system does not support the OMG IDL type `fixed`, which is a representation of binary-coded-decimal, or BCD. We have no immediate plans to support it in the future.

3.3 Access to Standard ILU Features

3.3.1 Servers and Ports

The C type which represents an ILU *kernel server* is `ILU_C_Server`. True servers can be created with the function `ILU_C_InitializeServer`.

Instances of `ILU_C_Server` are reference-counted, in the same way that instances of `ILU_C_OBJECT` are. Call `ILU_C_Server_duplicate` and `ILU_C_Server_release` to keep track of your usage of `ILU_C_Server` values. `ILU_C_CloseServer` and its relatives perform functions orthogonal to `ILU_C_Server_release`. The actual server will be destroyed only when there is no reason to maintain it; in addition to the reference counting introduced here, the existence of `ILU_C_OBJECT` values in that server count as another reason to maintain the server.

A kernel server can export itself via multiple ports. `ILU_C_InitializeServer` may create one, and `ILU_C_AddPort` can be called to create more.

3.3.1.1 Object Tables

It is sometimes useful to have a server create true objects only when they are mentioned by a client's actual invocation of a method on them. This is allowed in ILU by an interface called an *object table*, associated with a server. An object table contains two functions, one of which creates a new true instance when called with an instance handle, and the other of which frees the object table when the server is destroyed. Object tables are associated with servers when the server is created. A value of type `ILU_C_ObjectTable` may be created by a call on `ILU_C_CreateObjectTable`; see the API reference for more information about this function.

3.3.1.2 Server Relocation

It is sometimes useful to have a 'dummy' server, that will redirect any requests to it to a real server somewhere else. This can be used for load balancing, automatic start-up of services, redirecting name service, code migration, and other various purposes. ILU supports this via a mechanism called *server relocation*, which can be used in C via the function `ILU_C_SetServerRelocationProc()`.

3.3.2 Threading and Event Loops

The ILU C runtime allows the use of several different kernel threads packages, application-specific threads packages, or various kinds of event loops. See Chapter 12 [Threads and Event Loops], page 191 for a general discussion of threads and event loops in ILU.

The two macros `ILU_C_USE_OS_THREADS` and `ILU_C_FINISH_MAIN_THREAD` (see Section 3.5.6 [Server Manipulation], page 67) are provided to allow use of standard Win32 threads, POSIX threads, or Solaris threads. Application-specific threads packages can also be used by explicitly setting the *wait tech*, *lock tech*, and *main loop* via calls on the ILU kernel (see ⟨undefined⟩ [], page ⟨undefined⟩ for more information on these functions), then instructing the C runtime on how to fork a new thread by calling `ILU_C_SetFork`.

If no thread technology is specified, ILU's C runtime will operate in a single-threaded (i.e., event loop) manner, using the default ILU main loop. That main loop can also be replaced with an application-specific event loop system if desired; this is often useful when using a toolkit like Xt or Tk. This replacement works by calling the kernel procedure `ilu_SetMainLoop`. In the case of the C runtime, moreover, several examples of how to set the event loop are provided in the directory '*ILUSRC*`/runtime/mainloop/`'. They will have been automatically built into a library and installed into '*ILUHOME*`/lib/`' if configured in during the build process. They are useful examples of how to set the event loop to override the ILU default event loop.

For single-threaded operation, the main loop must be invoked. This can be done with either ILU_C_Stoppable_Run, ILU_C_Run, ILU_C_StoppableRun, or ILU_C_FINISH_MAIN_THREAD (see Section 3.5.6 [Server Manipulation], page 67 --- which is woefully incomplete and out of date).

For multi-threaded operation, no main loop need --- or even really can --- be invoked. Some of the aforementioned "main loop" procedures also "work" in multi-threaded runtimes --- they simply block the calling thread.

3.3.3 Custom Records

Custom record support in the ILU C runtime provides support for replacing the default generated record type R with a different struct type S. S must have a field of type R as its first field.

Custom record support for a particular ISL record type can be specified by calling ILU_C_RegisterCustomRecord() on the CORBA_TypeCode value for the type. Note that this allows you to register a function to be called just before marshalling the value to another address space (among other functions). This 'pre-output' function may be called more than once on the same call, if the protocol selected requires sizing of arguments. It should be written so that repeated calls on the same value have no effect.

3.3.4 Custom Surrogates

Custom surrogates allow the user to specify custom surrogate object types which may have additional functionality in terms of caching or other side effects, and have them created instead of the default ILU surrogate object type when an instance is received. This functionality is provided in the C runtime with the function ILU_C_RegisterCustomSurrogateType.

3.3.5 String Binding Handle Formation

When a client program is making a call on an object which is provided via an object table, it is often convenient to allow the client to form an appropriate string binding handle for the object, then instantiate a surrogate instance from that string binding handle. The function ILU_C_FormSBH is provided to aid client programs in forming string binding handles.

3.3.6 Simple Binding

The ILU C runtime provides the standard interface to ILU's *simple binding* mechanism. The function ILU_C_PublishObject publishes a true instance; ILU_C_WithdrawObject allows it to be withdrawn from the registry. The function ILU_C_LookupObject finds and returns an instance with the specified object ID.

3.3.7 Identities and Passports

See ILU_C_CreatePassport, ILU_C_AddIdentity, and ILU_C_DestroyPassport for managing ilu_Passports.

To pass/receive an `ilu_Passport` in a call, the caller calls `ILU_C_SetPassportContext(`*pp*`)` to store *pp* in a special hidden per-thread slot, and `ILU_C_SetPassportContext()` to retrieve it. *pp* will remain in that slot until another call to `ILU_C_SetPassportContext` overwrites it. The callee calls `ILU_C_CallerIdentity()` in a true method to fetch *pp* from the slot. If the caller and callee are both written in C, and the true method has been invoked directly from the same thread, the returned value will be whatever passport has been set with `ILU_C_SetPassportContext()`; otherwise it will be the passport passed by the caller.

3.3.8 Pipelining in C

A pipeline (see Section 1.3.4 [Pipelining], page 9) is represented in C by an `ILU_C_Pipeline`, and is created by calling `ILU_C_CreatePipeline`. A pipeline is associated with calls via a special hidden per-thread slot; this slot is accessed with `ILU_C_SetPipelineContext` and `ILU_C_GetPipelineContext`. When the `ILU_C_Pipeline` is no longer needed, the client calls `ILU_C_ReleasePipeline`.

3.3.9 Call Order Preservation in C

A serializer (see Section 1.3.5 [Call Order Preservation], page 10) is represented in C by an `ILU_C_Serializer`, and is created by calling `ILU_C_CreateSerializationContext`. A serializer is associated with calls via a special hidden per-thread slot; this slot is accessed with `ILU_C_SetSerializationContext` and `ILU_C_GetSerializationContext`. When the `ILU_C_Serializer` is no longer needed, the client calls `ILU_C_ReleaseSerializer`.

3.3.10 Batching in C

A batcher (see Section 1.3.6 [Batching], page 11) is represented in C by an `ILU_C_Batcher`, and is created by calling `ILU_C_CreateBatcher`. A batcher is associated with calls via a special hidden per-thread slot; this slot is accessed with `ILU_C_SetBatcherContext` and `ILU_C_GetBatcherContext`. The application can call `_C_PushBatcher` to initiate delivery all of all call messages buffered for a given batcher. When the `ILU_C_Batcher` is no longer needed, the client calls `ILU_C_ReleaseBatcher`.

3.4 Building C/ILU Applications

3.4.1 Using the ILU C Stubber

To generate C stubs from an ISL file, use the program c-stubber. Four files are generated from the ‘`.isl`’ file:

- ‘*interface-name*`.h`’ contains the definitions for the types and procedures defined by the interface and used by the generated stubs.
- ‘*interface-name*`-common.c`’ contains the general code used by both client and server; and
- ‘*interface-name*`-surrogate.c`’ contains the client-side and general code for the interface; and

- '*interface-name*-`true.c`' contains the server-side stubs and code for the interface.

Typically, clients of a module never have a need for the '*interface-name*-`true.c`' file.

```
% c-stubber foo.isl
header file for interface foo to ./foo.h...
common code for interface foo to ./foo-common.c...
code for surrogate stubs of interface foo to ./foo-surrogate.c...
code for true stubs of interface foo to ./foo-true.c...
%
```

3.4.1.1 Command-line Options

The program `c-stubber` supports the following options:

- `-I` *directory* -- add *directory* to the list of directories to search for interface definition files. Note that the *directory* must be separated from the `-I` with whitespace, unlike the convention for C compilers.
- `-dir` *directory* -- put output files in *directory*. Will attempt to create *directory* with "`mkdir` *directory*" if not already present.
- `-true` -- generate true code.
- `-tname` *filename* -- put true code in file called *filename*.
- `-common` -- generate common code.
- `-cname` *filename* -- put common code in file called *filename*.
- `-surrogate` -- generate surrogate code.
- `-sname` *filename* -- put surrogate code in file called *filename*.
- `-headers` -- generate header code.
- `-hname` *filename* -- put header code in file called *filename*.
- `-removefirst` -- for generated files, remove file before generating a new version of the file.
- `-nodefaulttrueclass` -- for true code, don't generate the commands which create default true classes for each class in the interface. This avoids pulling in references to possibly unused default true methods.
- `-nodefaulttrueclassfor` *classname* -- for true code, don't generate the commands which creates a default true class for the class *classname*. This avoids pulling in references to possibly unused default true methods. *classname* is specified as *INTERFACENAME*.*TYPENAME*, using the ISL interface name and type name.
- `-renames` *filename* -- use the specified renaming file *renames-file*. See the following section on ''Tailoring Identifier Names'' for more information on this.

If none of `-true`, `-surrogate`, `-common`, `-headers` is specified, the default action is to produce all of them. However, if any of those switches is explicitly specified, only those specified will be produced.

3.4.1.2 Tailoring Identifier Names

The option `-renames` *renames-filename* may be used with `c-stubber` to specify particular C names for ISL types.

It is sometimes necessary to have the C names of an ILU interface match some other naming scheme. A mechanism is provided to allow the programmer to specify the names of C language artifacts directly, and thus override the automatic ISL to C name mappings.

To do this, you place a set of synonyms for ISL names in a *renames-file*, and invoke the `c-stubber` program with the switch `-renames`, specifying the name of the renames-file. The lines in the file are of the form

construct ISL-name C-name

where *construct* is one of `method`, `exception`, `type`, `interface`, or `constant`; *ISL-name* is the name of the construct, expressed either as the simple name, for interface names, the concatenation *interface-name*`.`*construct-name* for exceptions, types, and constants, or *interface-name*`.`*type-name*`.`*method-name* for methods; and *C-name* is the name the construct should have in the generated C code. For example:

```
# change "foo_r1" to plain "R1"
type foo_r1 r1
# change name of method "m1" to "method1"
method foo_o1_m1 method1
```

Lines beginning with the 'sharp' character '`#`' are treated as comment lines, and ignored, in the renames-file.

This feature of the `c-stubber` should be used as little and as carefully as possible, as it can cause confusion for readers of the ISL interface, in trying to follow the C code. It can also create name conflicts between different modules, unless names are carefully chosen.

3.4.2 Using ILU Modules From C

Before manipulating surrogate objects, a client module must first call a runtime initialization procedure *Foo*__`Initialize` for each ISL interface *Foo* that declares object types whose surrogates are to be manipulated. Additionally, server modules must also call server initialization procedures (see previous section). These initialization calls may be made in any order, and each procedure may be called more than once. However, no two calls may be done concurrently (this is an issue only for those using some sort of multi-threading package).

A client of an exported module may obtain an object instance either by calling a method which returns the instance, or by calling *TYPE*__`CreateFromSBH()` on the string binding handle of an instance. Once the object instance, which is typically a surrogate instance, but may in fact be a true instance, is held by the client, it can be used simply by making method calls on it, as shown above.

3.4.3 Implementing an ILU Module With C

This section will outline the construction of a true module exported by an address space. For the example, we will demonstrate the calculator interface described above. We will also use the CORBA 2.0 names for standard types and exceptions, to show that it can be done.

First, some runtime initialization of the server stubs must be done. Call *Foo*`__InitializeServer` for every ISL interface *Foo* containing an object type implemented by the address space. Due to a misfeature in the current C support, also call *Bar*`__InitializeServer` for every ISL interface *Bar* containing an object type that is a supertype of one defined in *Foo* (if you don't, the server will get a runtime fault --- due to calling through a NULL procedure pointer --- when serving a call on an inherited method); this may cause you to have to supply dummy procedures, as explained in Section 3.2.8.2 [Interface Inheritance], page 42. Also call any client initialization procedures needed (see next section). These server and client initialization calls can be made in any order, and each initialization procedure can be called more than once. However, no two calls may be done concurrently (this is an issue only for those using some sort of multi-threading package).

Then we create an instance of `calc_self`. We then make the string binding handle of the object available by printing it to stdout. Finally the `ILU_C_Run` procedure is called. This procedure listens for connections and dispatches server methods.

The main program for the server is as follows:

```
#include "I2.h"

CORBA_long
  server_calc_self_Div (calc_self h,
                        CORBA_long u,
                        CORBA_long v,
                        CORBA_Environment *s)
{
  calc_numerator n = 9;

  if ( v == 0 )
    {
      calc__BindExceptionValue(s, ex_calc_DivideByZero, n);
      return( u );
    }
  else
    return( u / v );
}

main (int ac, char **av)
{
  calc_self s;
  char * sbh;
  CORBA_Object the_orb;
  CORBA_Environment ev;
```

```
    the_orb = CORBA_ORB_init (&ac, av, "ilu", &ev);
    if (!ILU_C_SUCCESSFUL(&ev)) {
      fprintf (stderr, "Can't initialize ORB; exception <%s>\n", CORBA_exception_id(&ev)
      CORBA_exception_free(&ev);
      exit(1);
    };

    calc__InitializeServer( );

    s = calc_self__CreateTrue (NULL, NULL, NULL);
    if (s == NULL)
      {
        fprintf (stderr, "Unable to create instance of calc_self.\n");
        exit(1);
      }
    else
      {
        sbh = CORBA_ORB_object_to_string (the_orb, s, &ev);
        if (ev._major == CORBA_NO_EXCEPTION)
          {
            printf ("%s\n", sbh);
            ILU_C_Run (); /* enter main loop; hang processing requests */
          }
        else
          {
            fprintf (stderr,
                     "Attempt to obtain sbh of object %p signalled <%s>.\n",
                     s, CORBA_exception_id(&ev));
            CORBA_exception_free(&ev);
            exit(1);
          }
      }
  }
```

3.4.4 Libraries and Linking

For clients of an ILU module, it is only necessary to link with the '*interface-name*-`surrogate.o`' and '*interface-name*-`common.o`' files generated from the C files generated for the interface or interfaces being used, and with the two libraries '*ILUHOME*`/lib/libilu-c.a`' and '*ILUHOME*`/lib/libilu.a`' (in this order, as '`libilu-c.a`' uses functions in '`libilu.a`').

For implementors of servers, the code for the server-side stubs, in the file '*interface-name*-`true.o`' compiled from '*interface-name*-`true.c`', and in the file '*interface-name*-`common.o`' compiled from '*interface-name*-`common.c`', should be included along with the other files and libraries.

3.5 ILU C API

In addition to the functions defined by the CORBA mapping, the ILU C mapping provides some other functions, chiefly for type manipulation, object manipulation, and server manipulation. There are also a number of macros provided for compatibility with both versions of CORBA (revision 2.0).

3.5.1 General

`CORBA_string` **ex_CORBA_InvalidName** Exception

Raised by `CORBA_ORB_resolve_initial_references` to indicate that no binding for the specified *service_name* is known. The associated value is the service name. Note that this differs from the strict CORBA definition of this exception, which has no associated value.

`CORBA_Object` **CORBA_ORB_init** (`int *` *argc*, `char **` *argv*, `CORBA_string` *orb_id*, `CORBA_Environment * env`) Function

Locking: Main Invariant holds

Called to initialize the ILU runtime, and acquire the ''orb'' object. The return value of this call is used in subsequent calls to other CORBA utility functions.

If the environment variable `ILU_COS_NAMING_IOR` is bound to a string binding handle for a `CosNaming` service, this call will bind the service name `"NameService"` to the object specified by that string binding handle.

`CORBA_ORB_ObjectIdList` **CORBA_ORB_list_initial_services** (`CORBA_Object` *the_orb*, `CORBA_Environment * env`) Function

Locking: Main Invariant holds

Returns a list of service names which can be usefully used in calls to `CORBA_ORB_resolve_initial_references()`. The type `CORBA_ORB_ObjectIdList` is a normal CORBA sequence of strings.

3.5.2 Type Manipulation

`OPTIONAL(ilu_Class)` **ILU_C_FindILUClassByTypeName** (`RETAIN(ilu_string)` *type-name*) Function

Locking: L1_sup < otmu, L2, Main unconstrained.

Given the *type-name* of an ILU object type, of the form `"Interface.Typename"`, returns the `ilu_Class` value for it. This value can be used to compare types for equality.

`OPTIONAL(ilu_Class)` **ILU_C_FindILUClassByTypeID** (`RETAIN(ilu_string)` *type-id*) Function

Locking: L1_sup < otmu; L2, Main unconstrained.

Given the *type-id* of an ILU object type, of the form `"ilu:gfbSCM7tsK9vVYjKfLole1HOBDc"`, returns the `ilu_Class` value for it. This value can be used to compare types for equality.

`GLOBAL(OPTIONAL(ilu_string))` **ILU_C_ClassName** (`RETAIN(CORBA_Object)`) — Function

Locking: unconstrained.

Returns the ILU name for the most specific type of an object instance.

`GLOBAL(OPTIONAL(ilu_string))` **ILU_C_ClassID** (`RETAIN(CORBA_Object)`) — Function

Locking: unconstrained.

Returns the ILU type ID for the most specific type of an object instance.

`ilu_Class` **ILU_C_ClassRecordOfInstance** (`CORBA_Object`) — Function

Locking: unconstrained.

Returns the `ilu_Class` value for the most specific type of an object instance.

`ILU_C_Class` **ILU_C_RegisterCustomSurrogateType** (`ilu_Class` *kernel-type*, `ILU_C_Class` *C-type*) — Function

Locking: unconstrained.

This function registers *C-type* as the kind of class to create an instance of when unmarshalling a surrogate instance of most specific type *kernel-type*. This should be used in conjunction with the automatically generated function *ObjectType*`__MakeClass()` (to create an instance of `ILU_C_Class`), and the automatically generated constant *ObjectType*`__MSType` (the appropriate value for *kernel-type*). This functionality can be used to implement application-specific surrogate types with caching and other extensions to the basic ILUmodel. The value returned is the previously registered surrogate class for this type.

`void` **ILU_C_RegisterCustomRecord** (`CORBA_TypeCode` *record_type*, `OPTIONAL(ILU_C_CRCreateFn)` *cfn*, `OPTIONAL(ILU_C_CRFreeFn)` *ffn*, `OPTIONAL(ILU_C_CRPreOutputFn)` *preoutfn*, `OPTIONAL(ILU_C_CRPostOutputFn)` *postoutfn*, `OPTIONAL(ILU_C_CRPostInputFn)` *postinfn*, `CORBA_Environment *` *env*) — Function

Locking: L1 < otmu

Registers a set of functions to be called for a particular record type *record_type*. The function *cfn* should create, initialize, and return a value of the desired type; it is called when the kernel is about to unmarshall a value of the type and needs storage for it. The function *freefn* should perform any clean-up operations necessary for the desired value; they are called before the standard free functions are called on the standard slots of the record type. The function *preoutfn* should prepare the value to be marshalled to another address space. It may be called repeatedly before the value is actually marshalled, due to sizing requirements imposed by various RPC protocols. The function *postoutfn* should perform any necessary cleanup after a value has been marshalled; it is expected that this will typically be a NIL function. The function *postinfn* should perform any initial setup needed after the standard slots of the value have been unmarshalled. This function may raise `BAD_PARAM` (if the type is not a record type), `NO_MEMORY`, and `INTERNAL`.

3.5.3 Object Manipulation

`ILU_C_Object *` **ILU_C_CreateTrueObject** (`ILU_C_Class` *c*, `OPTIONAL(ilu_string)` *instance_handle*, `OPTIONAL(ILU_C_Server)` *server*, `void * ` *instanceData*, `ilu_boolean` *inside_server*) Function

Locking: *inside_server* ? Inside(*server*, most specific type implemented by *c*) : Main Invariant.

Instantiates a true object of class *c*, in server *server*. If the *instance_handle* is `NIL`, one will be invented. If the *server* is `NIL`, the default server will be used. The *instanceData* is for the private use of the methods of the class. *inside_server* is `FALSE` for normal cases, but `TRUE` for use in the implementation of an object table.

`(OPTIONAL(ILU_C_Object *))` `ILU_C_FindObject` (`ilu_string` *server-id*, `ilu_string` *instance-handle*) Function

Locking: Main invariant holds.

Given the *server-id* and *instance-handle* of an object, returns the object if it exists in the current address space, or `ILU_NIL` if it doesn't exist. Unlike `ILU_C_SBHToObject()` and `ILU_C_LookupObject()`, this function will not create a surrogate for an instance if does not exist -- *but* if the *server-id* indicates a server with an object table, the *server* may create the object dynamically.

`ilu_boolean` **ILU_C_IDOfObject** (`CORBA_Object` *instance*, `PASS(char **)` *server-id*, `PASS(char **)` *instance-handle*) Function

Locking: Main invariant holds.

Given an *instance*, returns the server ID and instance handle of that instance. The strings returned are copies; the user must free them with `ilu_free` when finished with them.

`ilu_string` **ILU_C_SBHOfObject** (`CORBA_Object` *instance*) Function

Locking: Main invariant holds.

Given an *instance*, returns a reference to that instance. The CORBA-specified routine `CORBA_ORB_object_to_string()` should typically be used instead.

`OPTIONAL(CORBA_Object)` **ILU_C_SBHToObject** (`char *` *sbh*, `ilu_Class` *static_type*, `RETAIN(CORBA_Environment *) Env`) Function

Locking: Main invariant holds.

Takes an object reference and returns the object. *static_type* is a type the caller knows the object to have.

`OPTIONAL(PASS(char *))` **ILU_C_FormSBH** (`RETAIN(char *)` *server-id*, `RETAIN(char *)` *instance-handle*, `ilu_Class` *most-specific-type*, `RETAIN(ilu_ProtocolInfo)` *pinfo*, `RETAIN(ilu_TransportInfo)` *tinfo*, `RETAIN(CORBA_Environment *) Env`) Function

Locking: Main invariant holds.

Takes necessary information about an object reference, and returns a well-formed ILU string binding handle for that information. This SBH can then be used to create a surrogate instance, using `ILU_C_SBHToObject`.

`OPTIONAL(PASS(char*))` **ILU_C_PublishObject** (`CORBA_Object` *instance*) Function

Locking: Main invariant holds.

Publishes the OID of the *instance* in a domain-wide registry. This is an experimental interface, and may change in the future.

`ilu_boolean` **ILU_C_WithdrawObject** (`CORBA_Object` *instance*, `PASS(char *)` *proof*) Function

Locking: Main invariant holds.

Removes the OID of the *instance* from the domain-wide registry. *proof* is the string returned from the call to `ILU_C_PublishObject()`.

`OPTIONAL(GLOBAL(CORBA_Object))` **ILU_C_LookupObject** (`RETAIN(char *)` *sid*, `RETAIN(char *)` *ih*, `ilu_Class` *static-class*) Function

Locking: Main invariant holds.

Using the local registry, find and return the object specified by the given Server ID and server-relative Instance Handle. *static_type* is one you know the actual object must have; it may also have more refined types. For an already-reified surrogate this procedure will reconsider what contact info to use for reaching the server.

`OPTIONAL(GLOBAL(CORBA_Object))` **ILU_C_CreateSurrogateObject** (`ilu_Class` *type*, `RETAIN(char *)` *ih*, `ilu_Server` *server*, `ILU_C_ENVIRONMENT` **env*) Function

Locking: Main invariant holds.

Create a new object instance of the specified *type* on the specified *server*, with the specified *ih*. If unable to create such an object, return `ILU_NIL`, and signal the error in *env*.

This procedure can be used to create new client-side objects for which no true object yet exists. This is the way a client using a server with an object table causes the server to create new instances 'on the fly'. When used in this way, the *ih* must contain all information necessary to allow the server to create the proper true object, as it is the only information passed to the object table's object creation procedure.

`CORBA_Object` **CORBA_Object_duplicate** (`CORBA_Object` *instance*, `CORBA_Environment *` *env*) Function

Locking: Main invariant holds.

Increments the reference count of the *instance*, and returns the instance.

`void` **CORBA_Object_release** (`CORBA_Object` *instance*, `CORBA_Environment *` *env*) Function

Locking: Main invariant holds.

Decrements the reference count of the *instance*. The instance may be destroyed as a result of this operation.

`CORBA_unsigned_long` **CORBA_Object_hash** (`CORBA_Object` *instance*, `CORBA_unsigned_long` *max_value*, `CORBA_Environment *` *env*) Function

Locking: Main invariant holds.

Returns a hash value for the *instance*, less than or equal to *max_value*. Mandated by the CORBA spec.

`CORBA_boolean` **CORBA_Object_is_equivalent** (`CORBA_Object` *instance1*, `CORBA_Object` *instance2*, `CORBA_Environment *` *env*) Function

Locking: Main invariant holds.

Returns `ilu_TRUE` if ILU believes the two instances to be the same object, `ilu_FALSE` otherwise. Compares the servers and kernel objects of the two instances.

`void` **ILU_C_PingObject** (`ILU_C_Object` *instance*, `ILU_C_ENVIRONMENT *` *env*) Function

Locking: Main invariant holds.

Attempts round-trip effectless call on object. May raise system exception to indicate failure.

`CORBA_boolean` **CORBA_Object_non_existent** (`CORBA_Object` *instance*, `CORBA_Environment *` *env*) Function

Locking: Main invariant holds.

Returns `ilu_TRUE` if object cannot be successfully pinged. May cause instance to be destroyed.

`CORBA_boolean` **CORBA_Object_is_a** (`CORBA_Object` *instance*, `CORBA_string` *repository_id*, `CORBA_Environment *` *env*) Function

Locking: Main invariant holds.

Returns `ilu_TRUE` if the *instance* supports the interface identified by *repository_id*. May involve a network round trip.

`CORBA_boolean` **CORBA_Object_is_nil** (`CORBA_Object` *instance*, `CORBA_Environment *` *env*) Function

Locking: Main invariant holds.

Returns `ilu_TRUE` if the *instance* is the `NIL` object reference.

`CORBA_string` **CORBA_ORB_object_to_string** (`CORBA_Object` *the_orb*, `CORBA_Object` *instance*, `CORBA_Environment *` *env*) Function

Locking: Main invariant holds.

Returns a string binding handle for the *instance*. The argument *the_orb* is the result of a call to `CORBA_ORB_init()`.

`CORBA_Object` **CORBA_ORB_string_to_object** (`CORBA_Object` *the_orb*, `CORBA_String` *string_binding_handle*, `CORBA_Environment *` *env*) Function

Locking: Main invariant holds.

Returns a `CORBA_Object` instance for the specified *string_binding_handle*. The argument *the_orb* is the result of a call to `CORBA_ORB_init()`.

`OPTIONAL(CORBA_Object)` **CORBA_ORB_resolve_initial_references** (`CORBA_Object` *the_orb*, `CORBA_String` *service_name*, `CORBA_Environment *` *env*) Function

Locking: Main invariant holds.

Returns an instance for the service named by *service_name*, if the system knows of one. The argument *the_orb* is the result of a call to `CORBA_ORB_init()`. Raises `ex_CORBA_InvalidName` if the *service_name* doesn't name a known service.

`ilu_integer` **ILU_C_SetObjectGCTimeout** (`ILU_C_Object *` *the_obj*, `ilu_integer` *timeout*, `ILU_C_ENVIRONMENT *` *env*) Function

Locking: Main invariant holds.

Sets the GC timeout of *the_obj* to *timeout*. Returns the previous timeout when successful. The GC timeout of an instance is the amount of time ILU will wait before collecting an object without references. This timeout accommodates references to the object that are in transit on the network, and its value should be related to typical maximum network delays on the network being used. Setting the timeout to a value less than the network delay may result in objects being prematurely collected.

3.5.4 Identity Functions

`ilu_Passport` **ILU_C_CreatePassport** Function
(`OPTIONAL(PASS(ilu_IdentityInfo)) info, ILU_ERRS((no_memory)) *err`)

Create and return a new passport object. If an identity *info* is passed in, will put that identity in the new passport.

`ilu_IdentityInfo` **ILU_C_CopyIdentity(RETAIN(ilu_IdentityInfo)** Function
`info, ILU_ERRS((no_memory)) *err`)

Creates and returns a copy of the identity in *info*.

`ilu_boolean` **ILU_C_AddIdentity(RETAIN(ilu_Passport)** `pp,` Function
`PASS(ilu_IdentityInfo) info, ILU_ERRS((no_memory)) *err`)

Adds the specified identity to the specified passport, which now owns the identity storage.

`OPTIONAL(RETAIN(ilu_IdentityInfo))` Function
ILU_C_FindIdentity(RETAIN(ilu_Passport) `pp, RETAIN(ilu_IdentityType) ident_type`)

If the passport *pp* contains an identity of the specified type, returns a pointer to it, otherwise returns `ILU_NIL`. The passport retains ownership of the identities storage; the caller may make a copy of the identity by calling `ILU_C_CopyIdentity`.

Several identity types are pre-defined. The identity type `ilu_ConnectionIdentity` is always defined; it consists of a string which describes the connection used by the caller to communicate with the server in a colloquial fashion. The identity type `ilu_SunRPCAuthUnixIdentity` is defined if the `Sun RPC` protocol has been configured in; it provides a struct containing the various pieces of information specified by the protocol specification. The identity type `ilu_GSSIdentity` is available if support for the secure transport has been configured in; it supports a variety of identity schemes under the GSS umbrella. See Chapter 14 [Security], page 217 and the file '*ILUSRC*`/runtime/kernel/iluxport.h`' for more information on identities and identity schemes.

`ilu_cardinal` **ILU_C_DisplayIdentity** (`RETAIN(ilu_IdentityInfo)` Function
`identity, RETAIN(char *) buf, ilu_cardinal bufsize, ILU_C_ENVIRONMENT * env`)

Formats a textual display of the *identity* into *buf*, respecting *bufsize*. Returns the actual length of the string (the amount of the buffer that was actually used). May raise an exception through *env*.

`ilu_boolean` **ILU_C_DecodeGSSIdentity** (`RETAIN(ilu_IdentityInfo) identity, OPTIONAL(gss_name_t *) name, OPTIONAL(ilu_FineTime *) good_till, OPTIONAL(gss_OID) mech, OPTIONAL(ilu_boolean) localp, OPTIONAL(OM_uint32 *) flags, ILU_C_ENVIRONMENT * env`) Function

Only available if ILU has been configured with support for the GSS security transport. Returns various aspects of the GSS identity in the output parameters *name*, *good_till*, *mech*, *localp*, and *flags*. If no return value is specified for an output parameter, that output parameter is not returned. May raise an exception through *env*, in which case the return value is `ilu_FALSE`. If no exception is signalled, the return value is `ilu_TRUE`.

The meaning of the output parameters are as follows:

- *name* -- the name of the principal in the identity.
- *good_till* -- the time when the identity expires.
- *mech* -- the specific security mechanism used to authenticate the identity. GSS is an 'umbrella' system which allows many different security mechanisms to be used through a standard interface.
- *localp* -- `ilu_TRUE` if the identity is local, `ilu_FALSE` if the identity has been established for a remote principal.
- *flags* -- this parameter is only valid for remote identities (*localp* => `ilu_FALSE`). These are the context flags returned from `gss_inquire_context` in the *ctx_flags* parameter. See '*ILUSRC*`/GSS/doc/draft-ietf-cat-gssv2-cbind-01.txt`' for a complete description.

See '*ILUSRC*`/GSS/kernel/gssapi.h`' for a definition of the GSS types `gss_name_t`, `gss_OID`, and `OM_uint32`.

`OPTIONAL(PASS(ilu_string))` **ILU_C_GSSNameToString** (`RETAIN(gss_name_t) gss_name, ILU_C_ENVIRONMENT * env`) Function

Returns a newly-malloced string containing a textual representation of the principal name in *gss_name*. May raise an exception through *env*, in which case returns `ILU_NIL`.

See '*ILUSRC*`/GSS/kernel/gssapi.h`' for a definition of the GSS type `gss_name_t`.

`ilu_IdentityInfo` **ILU_C_AcquireGSSIdentity** (`gss_cred_id_t` *c*, `ILU_C_ENVIRONMENT *` *err*) Function

Available only if the security transport filter has been configured in. Returns an `ilu_IdentityInfo` corresponding to the given GSS credentials. May raise an exception through *err*, in which case returns `ILU_NIL`.

See '*ILUSRC*`/GSS/kernel/gssapi.h`' for a definition of the GSS type `gss_cred_id_t`.

`OPTIONAL(gss_cred_id_t)` **ILU_C_AcquireGSSCredForName** (`char` **name*, `ilu_cardinal` *lifetime*, `gss_OID` *secmech*, `ilu_boolean` *accept_only*, `ILU_C_ENVIRONMENT *` *err*) Function

Available only if the security transport filter has been configured in. Returns GSS credentials given the cannonical inputs, which are

- *name* -- specifies the name of the principal to acquire credentials for. It should be of the form `<namespace-identifier>:<principal-name>`, where the namespace-identifier is a stringified `gss_OID`, and the principal-name is a name in that namespace's string formulation. An example would be `"1.2.840.113550.9.1.4:someone@parc.xerox.com"`. The namespace-identifier identifies the iso.member-body.US.Xerox.ILU.GSS.rfc822-namespace namespace, which supports RFC 822 style mail addresses for principal names. See Chapter 14 [Security], page 217 for a discussion of available namespaces.
- *lifetime* -- specifies the time in seconds that these credentials should be good for. The value `GSS_C_INDEFINITE` may be specified for an infinite period.
- *sechmech* -- a GSS OID specifying the particular security mechanism which should be used to validate the identity. See Chapter 14 [Security], page 217 for a discussion of available security mechanisms.
- *accept_only* -- specifies whether the credentials may be used to accept security contexts or initiate them. When specified as `ilu_TRUE`, the returned credentials may only be used to accept security contexts. When specified as `ilu_FALSE`, they may only be used to initiate security contexts.

May raise an exception through *err*, in which case a value of `ILU_NIL` is returned.

See '*ILUSRC*`/GSS/kernel/gssapi.h`' for a definition of the GSS types `gss_OID` and `gss_cred_id_t`.

`ilu_IdentityInfo` **ILU_C_AcquireSunRPCAuthUnixIdentity** (`ilu_string` *hostname*, `ilu_shortcardinal` *uid*, `ilu_shortcardinal` *gid*, `ilu_shortcardinal` *ngroups*, `ilu_shortcardinal*` *groups*, `ILU_C_ENVIRONMENT *` *env*) Function

Available only if SunRPC UNIX Authorization has been configured in. Returns an `ilu_IdentityInfo` corresponding to the given UNIX credentials.

`ilu_boolean` **ILU_C_SetPassportContext** (`ilu_Passport` *pp*) Function

Sets the special hidden per-thread slot for passports to contain *pp*. The slot retains that value until explicitly changed later.

`ilu_Passport` **ILU_C_GetPassportContext** (`void`) Function

Returns the value in the special hidden per-thread slot for `ilu_Passports`.

`ilu_boolean` **ILU_C_DestroyPassport** (`PASS(ilu_Passport) pp, ilu_Error * err`) Function

Deallocates the storage associated with the passport, and any associated identities.

`OPTIONAL(ilu_Passport)` **ILU_C_CallerIdentity** (`void`) Function

Returns the passport associated with the caller, or possibly `ILU_NIL` if being invoked directly in a thread with no passport set. This procedure should only be invoked inside the scope of a true method.

3.5.5 Call Management Functions in C

`ILU_C_Serializer` **ILU_C_CreateSerializationContext** (`ILU_C_Server` *S*, `ILU_C_ENVIRONMENT *`*env*) Function

Creates a new instance of the serialization guarantee; this instance is applicable only to calls on objects of *S*.

`ilu_boolean` **ILU_C_ReleaseSerializer** (`ILU_C_Serializer` *si*, `ILU_C_ENVIRONMENT *`*env*) Function

A client calls this after it is done using the given `ilu_Serializer`.

`ilu_boolean` **ILU_C_SetSerializationContext** (`ILU_C_Serializer` *x*) Function

Sets the special hidden per-thread slot for `ILU_C_Serializer`s to contain *x*. The slot retains that value until explicitly changed later.

`ILU_C_Serializer` **ILU_C_GetSerializationContext** (`void`) Function

Returns the value in the special hidden per-thread slot for `ILU_C_Serializer`s.

`ILU_C_Batcher` **ILU_C_CreateBatcher** (`ilu_FineTime` *timeout*, `ilu_boolean` *pushable*, `ILU_C_ENVIRONMENT *`*env*) Function

Creates a new batcher.

`ilu_boolean` **ILU_C_ReleaseBatcher** (`ILU_C_Batcher` *val*, `ILU_C_ENVIRONMENT *`*env*) Function

A client calls this after it is done using the given `ILU_C_Batcher`.

`ilu_boolean` **ILU_C_SetBatcherContext** (`ILU_C_Batcher` *x*) Function

Sets the special hidden per-thread slot for `ILU_C_Batcher`s to contain *x*. The slot retains that value until explicitly changed later.

`ILU_C_Batcher` **ILU_C_GetBatcherContext** (`void`) Function

Returns the value in the special hidden per-thread slot for `ILU_C_Batcher`s.

`ilu_boolean` **ILU_C_PushBatcher** (`ILU_C_Batcher` *b*, `ILU_C_ENVIRONMENT *` *env*) — Function

Initiates delivery of all buffered call messages associated with *b*.

`ILU_C_Pipeline` **ILU_C_CreatePipeline** (`ILU_C_ENVIRONMENT *`*env*) — Function

Creates a new pipeline.

`ilu_boolean` **ILU_C_ReleasePipeline** (`ILU_C_Pipeline` *pl*, `ILU_C_ENVIRONMENT *`*env*) — Function

A client calls this after it is done using the given `ILU_C_Pipeline`.

`ilu_boolean` **ILU_C_SetPipelineContext** (`ILU_C_Pipeline` *x*) — Function

Sets the special hidden per-thread slot for `ILU_C_Pipelines` to contain *x*. The slot retains that value until explicitly changed later.

`ILU_C_Pipeline` **ILU_C_GetPipelineContext** (`void`) — Function

Returns the value in the special hidden per-thread slot for `ILU_C_Pipelines`.

3.5.6 Server Manipulation

`ilu_boolean` **ILU_C_USE_OS_THREADS** — Macro Function

Locking: Main invariant holds.

This macro expands to a function call. If ILU has been configured with os-level thread support, calling this routine will 'turn on' that thread support for use with C. This means that a new thread will be forked to handle each incoming connection, in servers, and if the wire protocol being used permits it, a thread will be forked to handle each incoming request. This routine returns FALSE, and emits an error message, if something goes wrong with enabling thread support. It must be called before making any other ILU calls, and before initializing any interfaces via calls to *interface*`__Initialize` or *interface*`__InitializeServer`.

`void` **ILU_C_FINISH_MAIN_THREAD** (`int` *returnvalue*) — Macro Function

Locking: Main invariant holds.

This routine will return from the 'main' thread with the specified value. In some thread systems, the program will be terminated when the main thread returns from `main()`, regardless of whether other threads are running. For these thread systems, this call will simply cause the main thread to idle forever, instead of returning.

`void` **ILU_C_Run** (`void`) — Function

Locking: Main invariant holds.

Called to animate a server and/or other parts of the program. Used only in single-threaded mode. Invokes the event handling loop. Never returns.

`OPTIONAL(ILU_C_Server)` **ILU_C_InitializeServer** Function
(`OPTIONAL(RETAIN(char *))` *serverID*, `OPTIONAL(GLOBAL(ILU_C_ObjectTable))` *obj_tab*, `OPTIONAL(RETAIN(ilu_ProtocolInfo))` *protocol*, `OPTIONAL(RETAIN(ilu_TransportInfo))` *transport*, `OPTIONAL(RETAIN(ilu_Passport))` *identity*, `ilu_boolean` *createPortAnyway*)

Locking: Main invariant holds.

Creates and returns an `ilu_Server` with ID *serverID*, object mapping table *obj_tab*, using protocol *protocol* over a transport stack specified by *transport*. If *serverID* is specified as `NULL`, a unique string is generated automatically for the server ID. If *obj_tab* is specified as `NULL`, the default hash table object table is used.

If either *protocol* or *transport* is specified, or if *createPortAnyway*, an `ilu_Port` will automatically be created and added to the `ilu_Server`. *protocol*, if not `NULL`, is a string that specifies which RPC protocol to use on the port; `NULL` causes use of the default protocol. *transport*, if not `NULL`, is a sequence of strings that specifies the transport stack to use below the RPC protocol; `NULL` signifies use of the default transport to/from one of the IP addresses of this host. See Section 13.2.5 [Protocols and Transports], page 216 for details on protocols and transports. If an *identity* is specified, it may be used for communications security purposes. If an `ilu_Port` is called for, it will become the default port of the `ilu_Server`.

`ilu_boolean` **ILU_C_AddPort** (`ILU_C_Server` *server*, Function
`OPTIONAL(RETAIN(ilu_ProtocolInfo))` *protocol*, `OPTIONAL(RETAIN(ilu_TransportInfo))` *transport*, `OPTIONAL(RETAIN(ilu_Passport))` *identity*, `ilu_boolean` *makeDefault*, `ILU_C_ENVIRONMENT *` *env*)

Locking: Main invariant holds.

Creates a new `ilu_Port` for the *server*. *protocol*, *transport*, and *identity* parameterize the `ilu_Port` as for `ILU_C_InitializeServer`.

`(RETAIN(ilu_string))` **ILU_C_IDOfServer** (`ILU_C_Server` *server*) Function

Locking: Main invariant holds.

Returns a pointer to the server id of the specified server.

`ILU_C_ObjectTable` **ILU_C_CreateObjectTable** (`CORBA_Object` Function
(`*`*object_of_ih*)(`ilu_string instance-handle, ilu_private user-data`), `void` (`*`*free_user_data*)(`ilu_private user-data`), `ilu_private` *user-data*)

Locking: Main invariant holds.

Locking for *object_of_ih*: L1 >= {server}, L1 >= {gcmu} if result is true and collectible; L2, Main unconstrained.

Locking for *free_user_data*: L1 >= {server}; L2, Main unconstrained.

Creates and returns a value of type `ILU_C_ObjectTable` encapsulating the two procedures *object_of_ih* and *free_user_data*, and the user-specified data element *user-data*. When *object_of_ih* is called, it should create an appropriate `CORBA_Object` with the specified instance handle, and return it. When *free_user_data* is called, it indicates the end of the object table, and *free_user_data* should free up any storage associated with *user-data*.

An object table is associated with a kernel server by passing the object table as a parameter to the function `ILU_C_InitializeServer`. A single object table may be used with multiple different `ilu_Server` instances.

ILU_C_ServerRelocateProc C Procedure Type

Locking: L1 >= {ilu_Server}; L2 unconstrained

```
typedef ilu_boolean
  (*ILU_C_ServerRelocateProc) (ILU_C_Server server,
       ilu_private argument,
       OPTIONAL(ilu_ProtocolInfo *) new_pinfo,
       OPTIONAL(ilu_TransportInfo *) new_tinfo);
```

This function should return TRUE if new pinfo and tinfo have been stored into the out parameters; otherwise it should return FALSE. It has no mechanism for signalling errors. It is called by a protocol implementation to see if the server wants the caller to be redirected to another location or cinfo stack.

`OPTIONAL(void *)` **ILU_C_SetServerRelocationProc** (`ILU_C_Server` *server*, `ILU_C_ServerRelocateProc` *relocation_fn*, `void *` *relocation_arg*, `ILU_C_ENVIRONMENT * metavarenv`) Function

Locking: Main invariant holds

Ensures that the function *relocation_fn* will be called with *relocation_arg* as an argument whenever a request comes in for an object maintained by *server*, on any connection which uses a relocating protocol (currently only `w3ng` is a relocating protocol). See the definition of `ILU_C_ServerRelocateProc` for a description of how it is used. The returned value is the previous value of *relocation_arg*, if any.

`ilu_cardinal` **ilu_tcp_SetDefaultBuffersize** (`ilu_cardinal` *new-buffer-size*) Function

Locking: Main invariant holds.

Sets the default buffersize, in bytes, for TCP transports to *new-buffer-size*. This default can be overridded by explicitly specifying a buffersize in the tinfo for the port or object. Returns the previous default value. This function is only available if ILU has been configured with support for the TCP/IP transport.

`void` **ilu_tcp_GetStats** (`ilu_cardinal *` *bytes-sent*, `ilu_cardinal *` *bytes-read*, `ilu_cardinal *` *moorings-created*, `ilu_cardinal *` *connections-accepted*, `ilu_cardinal *` *connections-opened*, `ilu_cardinal *` *currently-open-connections*, `ilu_cardinal *` *max-simultaneously-open-connections*) Function

Locking: Main invariant holds.

Returns various statistics about the TCP/IP tranports use of various resources for this process. The values returned are the values since the process was started, or since the reset function `ilu_tcp_InitializeStats()` was last called, except for *currently-open-connections*, which is unaffected by the reset function. This function is only available if ILU has been configured with support for the TCP/IP transport.

`void` **ilu_tcp_InitializeStats** () Function

Locking: Main invariant holds.

Resets the statistics counters for this process. This function is only available if ILU has been configured with support for the TCP/IP transport.

`ilu_FineTime` **ILU_C_SetDefaultGCPingPeriod** (`ilu_FineTime` *new_period*, `ILU_C_ENVIRONMENT *` *env*) Function

Locking: L1 < gcmu

The ILU distributed garbage collection protocol detects defunct clients by periodically pinging their GC callback objects. If a client's callback object cannot be successfully pinged, it is removed from the list of clients which have references to any objects on the server. This call sets the ping period to *new_period*. Only GC callback objects registered after this call will use the new period. Returns the previous ping period upon success.

3.5.7 CORBA Compatibility Macros

ILU supports CORBA 2.0, and formerly supported either 1.1 or 1.2, depending on how it was installed at your site. A number of macros are defined to make programs less dependent on which version they use.

ILU_C_OBJECT Macro

Expands to `CORBA_Object`.

ILU_C_ENVIRONMENT Macro

Expands to `CORBA_Environment`.

ILU_C_NO_EXCEPTION Macro

Expands to `CORBA_NO_EXCEPTION`.

ILU_C_USER_EXCEPTION Macro

Expands to `CORBA_USER_EXCEPTION`.

ILU_C_SYSTEM_EXCEPTION Macro

Expands to `CORBA_SYSTEM_EXCEPTION`.

ILU_C_SUCCESSFUL (*ILU_C_ENVIRONMENT * ev*) Macro

Evaluates to true if no exception has been raised.

ILU_C_SET_SUCCESSFUL (*ILU_C_ENVIRONMENT * ev*) Macro

Sets *ev* to a successful result.

ILU_C_EXCEPTION_ID (*ILU_C_ENVIRONMENT * ev*) Macro

Returns the `char *` value that is the exception's ID.

ILU_C_EXCEPTION_VALUE (*ILU_C_ENVIRONMENT * ev*) Macro

Expands to `CORBA_exception_value(`*ev*`)`.

ILU_C_EXCEPTION_FREE (*ILU_C_ENVIRONMENT * ev*) Macro

Expands to `CORBA_exception_free(`*ev*`)`.

4 Using ILU with Python

4.1 Introduction

This document is for the Python programmer who wishes to use ILU. The following sections will show how ILU is mapped into Python constructs and how both Python clients and servers are generated and built.

4.2 The ISL Mapping to Python

4.2.1 Names

In general, ILU constructs Python symbols from ISL names by replacing hyphens with underscores. For example, an ISL object type `T-1` would correspond to the Python class `T_1`. Any place an ISL name appears as part or all of a Python identifier, this translation occurs.

4.2.2 Interface

Each ISL `interface` *I* generates two Python modules: one named *I* containing common definitions, and another named *I*`__skel` containing skeletons (server stubs). For example, `INTERFACE map-test;` generates the Python modules `map_test` and `map_test__skel`, contained in the files '`map_test.py`' and '`map_test__skel.py`', respectively.

4.2.3 Basic Types

The basic ISL types have the following mapping to Python types:

1. BYTE, BOOLEAN, SHORT CHARACTER, CHARACTER, SHORT INTEGER, INTEGER, and SHORT CARDINAL all map to Python int.
2. LONG INTEGER, CARDINAL, and LONG CARDINAL all map to Python long int.
3. SHORT REAL and REAL map to Python float.
4. LONG REAL maps to the Python type ilu_longreal, a type implemented by the ILU Python runtime. This type has limited functionality, but can be passed around without loss of precision, converted to float or int, and compared. A value of this type may be constructed by calling `ilu.LongReal()`.

4.2.3.1 Constant

ISL constants translate to Python variables initialized to the specified value. For example,

```
CONSTANT pi : real = 3.14159265358979323846;
```

maps to

```
pi = 3.14159265358979323846e0
```

4.2.4 Strings

An ISL `SEQUENCE OF SHORT CHARACTER` maps into a Python string. `SEQUENCE OF BYTE` is also mapped into a Python string.

4.2.5 Pickles and Typecodes

A value corresponding to the ISL type `PICKLE` is an instance of the Python class `ilu.Pickle`. Instances of this class have the following methods:

- `typecode()` - returns the typecode of the pickle's value as a string.
- `value()` - returns the Python form of the value in the pickle.
- `bytes()` - the pickled bytes of the pickled value as a string.

The constructor for this class takes two arguments, *typecode* and *value*, and returns a new pickle containing the value specified by *value* of the ISL type specified by *typecode*. Pickles may also be created by calling the constructor with a single argument string, which must be the result of an earlier call on the `bytes()` method of another pickle instance.

Typecodes are represented by the Python class `ilu.Typecode`. Typecodes are constructed with a single string argument, of the form `'`*interface*`.`*type*`'`, where *interface* is the ISL name for the interface, and *type* is the ISL name for the type. Instances of the `Typecode` class support the method

- `id()` - return the ILU type ID (CORBA repository ID) for the typecode's type.
- `name()` - return the ISL name of the typecode's type.

4.2.6 Constructed Types

4.2.6.1 Enumeration

Enumerations are mapped to a method-less class object which contains an attribute of the correct type and value for each value in the enumeration. The class also contains a dictionary, called ''__image__'', for each enumeration type that maps an enumeration value to a string corresponding to its Python enumeration value name.

For example,

```
TYPE color = ENUMERATION red, dark-blue END;
```

maps to

```
class color:
    red = 0;
    dark_blue = 1;
    __image__ = {
       red: 'red',
```

```
        dark_blue: 'dark_blue'};
```

4.2.6.2 Array

An ISL array maps into a Python list with the specified number of elements. Tuples as well as lists are accepted as input, but lists are always produced as output from ILU. Arrays of BYTE or SHORT CHARACTER are represented with Python strings.

4.2.6.3 Sequence

Sequences of BYTE or SHORT CHARACTER are represented as Python strings.

If your ILU system has been configured with *--enable-python-dictionaries*, sequence types matching a particular profile will be mapped to Python dictionaries. The sequence type must have a name that ends with either `"dict"` or `"Dict"`; the base type of the sequence type must be a record type; the record type must have exactly two fields; the two fields must be named `name` and `value`, in that order; and the type of the `name` field must be either an integer, byte, string, or cardinal type.

All other ISL sequence types map into Python lists. Tuples as well as lists are accepted as input, but lists are always produced as output from ILU.

4.2.6.4 Record

ISL records map into generated Python classes with the same name, with the record's field names as attributes. The name of the record becomes a constructor function which accepts exactly the same number of arguments as the record has fields, in the same order.

For example, a record value of the ISL type

```
TYPE segment = RECORD left-limit : integer, right-limit : integer END;
```

with a left-limit of -3 and a right-limit of 7 would map to

```
segment(-3, 7) => &lt;segment:{'left-limit' : -3, 'right-limit' : 7}&gt;
```

4.2.6.5 Union

An ISL union maps into a Python tuple with two components: an integer discriminator, and the discriminated value. There are three possibilities:

1. If the discriminator matches one of the union case values of an arm, the second component is of the type specified by that arm.
2. If the discriminator matches no union case values and there is a default arm, the second component is of the type specified by the default arm.
3. If the discriminator matches no union case values and there is no default arm but the union has the `OTHERS` attribute, the second component is `None`.

4.2.6.6 Optional

A value corresponding to the ISL type `OPTIONAL` *T* may be `None` (indicating the null case) in addition to the values of the type *T*.

4.2.7 Object Types

Each ISL object type is mapped into a Python class. These classes have the methods specified in the ISL, as well as some built-ins.

4.2.7.1 Surrogate and True Object Types

All surrogate object types inherit from `iluRt.IluObjSurr`, which in turn inherits from `iluRt.IluObject`. True object types inherit from `IluRt.IluObjTrue`, which also inherits from `iluRt.IluObject`. The method `IluTrueP()` will return a true value on true instances, and a false value on surrogate instances. The string binding handle of an object instance can be retrieved with the method `IluSBH()`. The object-id of an instance can be retrieved with `IluObjectID()`; it returns a tuple containing a string server ID and a string instance-handle. If support for the CORBA `IIOP` is configured into your ILU build, the string IOR of an instance can be retrieved by calling the function `ilu.IOROfObject()`, passing the instance as the argument. The type name of the most specific type of an instance can be retrieved with the method `IluTypeName()`; the unique ID of that type can be retrieve with the method `IluTypeID()`.

Object types which inherit from the ISL type `ilu.CORBA-Object` (which include all object types defined with OMG IDL), will inherit from the Python class `ilu.CORBA_Object`, which is the same as the class `CORBA.Object`.

4.2.7.2 Methods, Parameters, and Exceptions

ISL methods of an object type map to Python methods of the corresponding class. `IN` and `INOUT` parameters appear in the Python method signature in the same order as they do in ISL.

Let us define a *result* value to be either a return value (corresponding to a method's return type) or an `INOUT` or `OUT` parameter. Result values are returned by the Python method as a tuple, with the return value (if present) appearing before any parameters. If the method has only one result value, then it is simply returned (i.e., a tuple of length one is *not* constructed to hold this value). If the method has no result values, then `None` is returned.

An ISL exception translates to a Python variable initialized with a string representing the exception. These variables are used in Python `raise` statements in object implementation code, and in `try ... except` statements in client code. For example, the declaration

```
EXCEPTION division-by-zero;
```

in the interface `map-test` maps to the following statement in '`map_test.py`':

```
division = 'map-test: division-by-zero'
```

`ASYNCHRONOUS` methods have no return values and raise no user-specified exceptions. They may return before the completion of the true method. `FUNCTIONAL` methods that have no parameters are cached so that a surrogate address space makes only one call to the true address space to retrieve the return value.

4.2.7.3 Garbage Collection and COLLECTIBLE

All instances of ILU object types are covered by the normal Python garbage collection; i.e., the application program must maintain a reference to the instance, or it will be garbage collected. With true instances of COLLECTIBLE object types, the ILU kernel will maintain an additional reference to the instance as long as it has registered clients using that instance.

4.3 Access to standard ILU features

4.3.1 Servers and Ports

Each object exported by an implementation must belong to a true server, an instance of the Python type `ilu_Server` which is implemented by the ILU runtime. An `ilu_Server` can be created by calling the function `ilu.CreateServer([serverID [, transport [, protocol [, objectTable]]]])`, which returns a value of type `ilu_Server`. If *serverID* is a string, it specifies the server ID; if it is `None`, one will be invented automatically. The *transport* argument is either a sequence of strings, chosen to be compatible with the *protocol*, or `None` to let it default. The *protocol* argument is either a string specifying a particular RPC protocol, or `None` to choose the default. Additional ports can be added to a server with the `addPort()` method, if an application needs to make it available with via multiple protocols or addresses.

The first time a true server is created, it becomes the default server. The default server is used for an exported object if a server is not otherwise specified. If an object is exported before any servers have been created, one will be created automatically using default parameters and a message to that effect will be written to stderr.

An object of type `ilu_Server` has the following methods:

- `id()` - returns the ILU server ID of the server.
- `addPort(`*TINFO*`,` *PINFO*`)` - adds another port to the server with the specified *TINFO* and *PINFO*.
- `createSerializer()` - creates and returns a serialization context.

4.3.1.1 Object Tables

The *objectTable* argument allows specification of a callback function for creating true instances on demand. The callback function should take one argument, a string, which is the instance handle of the instance to be created, and return a true instance.

4.3.1.2 Server Relocation

It is sometimes useful to have a 'dummy' server, that will redirect any requests to it to a real server somewhere else. This can be used for load balancing, automatic start-up of services, redirecting name service, code migration, and other various purposes. ILU supports this via a mechanism called *server relocation*, which can be used in Python via the `setRelocator` method on the `ilu_Server` class.

4.3.2 Threading and Event Loops

To use threads, you must have configured both ILU and Python with thread support when building them. If you have done this, your ILU/Python runtime support will be thread-capable. To have ILU begin using threads, place a call to the function `ilu.ThreadedOperation()` in your Python program before any other ILU calls are made.

4.3.2.1 Animating Servers

Running the ILU main loop by calling `ilu.RunMainLoop()` brings the true servers to life. This function does not return until `ilu.ExitMainLoop()` is called. If you are using ILU with `Tkinter`, you should import `ilu_tk` before creating a loop handle, or calling `RunMainLoop`. `ilu_tk` sets things up so that both Tk and ILU events are handled.

4.3.2.2 Using Alarms

In order to schedule a Python function to be called at a certain time in the future when executing the ILU main loop, an `ilu_Alarm` may be used. Objects of this type are created by calling `ilu.CreateAlarm()`. An `ilu_Alarm` must be set to have any effect.

The alarm's method `set(time, proc, args)` is used to set the alarm. The `int`, `float`, or `ilu_FineTime` `time` argument is the time at which the alarm will fire; the `proc` argument is the Python function that will be called when the alarm fires; and the `args` argument is a tuple of arguments to be passed to `proc`. The tuple `args` must match `proc`'s signature. For example, if `proc` is declared `def P(a, b):` then `args` must be a two-tuple. Likewise, if `proc` takes only one argument then `args` must be a one-tuple, or if no arguments then a zero-tuple.

The function `ilu.FineTime_Now()` may be called to obtain ILU's idea of the current time. A value `sec` of type `int` or `float` in units of seconds may be converted to type `ilu_FineTime` by calling `ilu.FineTime(sec)`. Values of type `ilu_FineTime` may be compared, added, and subtracted. These operations may be used to construct values representing any relative time (subject to precision and range limitations), which is what is needed by an alarm's `set` method.

The alarm may be set multiple times with different arguments, in which case the parameters of the most recent call to `set` are in effect. Thus, once an alarm fires, it may be reused by calling `set` again.

An alarm may be unset by calling its method `unset()`.

4.3.3 Custom Records

ILU generally supports a facility named *custom records*. This means that an application can declare that the language-specific mapping of a particular record type *ISL(A)* to *lang(A)* is to be overridden, and that instead a specific type *X* will be used in this language to represent values of *ISL(A)*. In Python, this is done by simply replacing the generated class definition with a different class definition.

For example, suppose we had the ISL record type

```
INTERFACE Ifc;
  ...
TYPE Foo = RECORD color : RGB-tuple, position : XY-pair END;
```

The normal mapping of `Ifc.Foo` to Python would be to a class called `Foo` with the following definition:

```
class Foo (iluRt.IluRecord):
    __ilu_type_name__ = 'Ifc.Foo'
    def __init__(self, _color, _position)
        self.color = _color;
        self.position = _position;

    def __getinitargs__(self):
      return (self.color, self.position)
```

To override this, define a new class in your application that has matching signatures for `__init__` and `__getinitargs__`, and a matching value for `__ilu_type_name__`. It must also inherit from `IluRt.IluRecord`. Then assign the class object for this new class to the symbol `Foo` in the Python module `Ifc`. So:

```
class MyFoo (iluRt.IluRecord):
    __ilu_type_name__ = 'Ifc.Foo'
    def __init__(self, _color, _position):
        self.color = _color
        self.position = _position
        self.some_other_attr = whatever_I_want
        call_some_other_code(self)

    def __getinitargs__(self):
        return self.color, self.position

    ...possible other methods...

Ifc.Foo = MyFoo
```

4.3.4 String Binding Handle Formation

To use object tables properly, it is usually necessary for a client program to create a surrogate instance for which the true instance does not yet exist. In Python, this is done by creating a string binding handle for the object, then calling `ilu.ObjectOfSBH()` on that SBH. String binding handles may be formed by calling the function `ilu.FormSBH()`.

4.3.5 Simple Binding

A true instance may be published with the simple binding service by calling its method `IluPublish()`. A true instance may be unpublished by calling its method `IluWithdraw()`.

A published ILU object may be obtained by calling `ilu.LookupObject(sid, ih, cl)`, where `sid` is object's server's server ID, `ih` is the object's instance handle, and `cl` is its class.

4.3.6 Principal Identities and Passports

An ILU passport (see Chapter 14 [Security], page 217) is represented in Python by an instance of the `ilu_Passport` object type. Instances of this type can be obtained by calling `ilu.CreatePassport()`. Please see the documentation of that function for more information on the abilities of this object type.

The passport of the caller may be obtained in the true method by calling the ILU runtime routine `ilu.CallerIdentity()`. The 'native' passport may be obtained by calling `ilu.GetPassport()`. In the case of a local call, these two passports may be the same object. Passports are thread-local; that is, an application may use a different passport in each thread.

4.4 Building Python/ILU Applications

4.4.1 Stub Generation

To generate Python stubs from an ISL file, use the program `python-stubber`. Two files are generated from each ISL `INTERFACE` *name*:

- '*name*`.py`' containing code for constants, exceptions, and types defined in the interface, and
- '*name*`__skel.py`' containing code for the skeletons (server stubs) for object types defined in the interface.

Alternatively, for surrogate-side use the stubber can be run automatically by a hook in the `ilu` module. Calling `ilu.AutoImport()` will establish the stubber as part of the normal Python `import` machinery, and will cause '`.isl`' and '`.idl`' files in directories on your `ILUPATH` environment variable path to be automatically stubbed and loaded.

4.4.1.1 Command-line Options

The program `python-stubber` supports the following options:

- `-I` *directory* -- add *directory* to the list of directories to search for interface definition files. Note that the *directory* must be separated from the `-I` with whitespace, unlike the convention for C compilers.

- `-dir` *directory* -- put output files in *directory*. Will attempt to create *directory* with "`mkdir` *directory*" if not already present.
- `-quiet` -- run without normal output to stderr.
- `-stub` -- generate the '*name*`.py`' file.
- `-skel` -- generate the '*name*`__skel.py`' file.
- `-removefirst` -- for generated files, remove file before generating a new version of the file.

If neither `-stub` nor `-skel` is specified, both files are produced. However, if either is explicitly specified, only those specified will be produced.

4.4.2 Implementing an ILU module in Python

A Python module that implements ILU objects of types defined in `INTERFACE` *I* also imports from *I*`__skel`. This gives access to the skeleton classes from which implementation classes inherit.

4.4.2.1 Implementation Inheritance

An implementation of object type *T* from interface *I* needs to inherit from the class *I*`__skel.`*T*. If there is inheritance in the ISL, and an implementation of a subtype wants to inherit from an implementation of a supertype, the skeleton class must be appear in the list of base types before the implementation class.

For example, objects for the ISL

```
INTERFACE j;

TYPE c1 = OBJECT METHODS one() END;
TYPE c2 = OBJECT METHODS two() END;
TYPE c3 = OBJECT SUPERTYPES c1, c2 END METHODS three() END;
```

could be implemented in Python by

```
import ilu, j, j__skel

class c1(j__skel.c1):
    def one(self):
        ...

class c2(j__skel.c2):
    def two(self):
        ...

class c3(j__skel.c3, c1, c2):
    def three(self):
        ...
```

In this case `c3`'s method `one` is implemented by `c1.one` and `c3`'s method `two` is implemented by `c2.two`.

4.4.2.2 Exporting Objects

An object can be exported in one of three ways:

1. The object's string binding handle may be obtained by calling its method `IluSBH()` and communicating this somehow to a client, who then turns the handle back into an object by calling `ilu.ObjectOfSBH(cl, sbh)`.
2. The object may be published using the simple binding service by calling its method `IluPublish()`. In order for this to be effective, the object must have a well-known object ID, or the object ID must be communicated to clients, so clients can know what to pass to `ilu.LookupObject`. The object ID is a function of the object's instance handle and its server's server ID.
3. The object may be returned by a method or passed back in a method's `INOUT` or `OUT` parameter.

An object's instance handle can be controlled by setting the instance variable `IluInstHandle` before the object is first exported. If this instance variable is not set, and instance handle will be invented automatically.

An object's server can be controlled by setting the instance or class variable `IluServer` to a value of type `ilu_Server`. The value of this variable at the time an object is first exported will be used as the server for that object. If such a variable is not set, the default server is used.

4.4.3 Using an ILU module in Python

The ILU runtime interface is in the Python module `ilu`. Python definitions for ISL `INTERFACE` *I* are in the Python module *I*. As with any other modules in Python, these modules are imported using the `import` statement.

A client program may create an ILU object in one of three ways:

1. Knowing the string binding handle `sbh` and class `cl` of an object, call `ilu.ObjectOfSBH(cl, sbh)` which returns an instance of that class. For example, to obtain an instance of ISL type `square` from `INTERFACE shapes` whose string binding handle is `sbh`, one would call `ilu.ObjectOfSBH(shapes.square, sbh)`.
2. Knowing the object ID `(sid, ih)` and class `cl` of an object that has been published using the simple binding service, call `ilu.LookupObject(sid, ih, cl)` which returns an instance of that class (or `None` if the lookup fails).
3. Receive an instance as a result value from a method call that returns an object type or has an object type as an `INOUT` or `OUT` parameter.

4.4.4 CORBA Support in Python

This release of ILU has several nods to an eventual CORBA mapping for the Python language. The Python `CORBA` module contains support for the classes `CORBA::ORB` and `CORBA::Object`, and the `CORBA::ORB_init()` function. See the Python/ILU API Reference for more information on these.

4.4.5 Freezing a Python Application Containing ILU

```
From: "Martin v. Loewis" <loewis@informatik.hu-berlin.de>
Subject: Freezing ILU
Date: Fri, 13 Mar 1998 07:54:30 PST

I currently try to freeze a Python application that uses ILU. With ILU
building libilupython, this is already simple. It would be even
simpler if ILU installed a file Setup in the library directory with
the contents

  iluPr -L/usr/ilu-2.0a12/lib -lilupython -lilu

(Of course, it could contain comments :-) This tells makesetup that
the iluPr module is available by linking ilupython and python to the
freezed image. With this installation, I can freeze my application
with the command line

  PYTHONPATH=/usr/ilu-2.0a12/lib python -O freeze/freeze.py
        -o outdir -e /usr/ilu-2.0a12/lib myscript.py

Of course, /usr/ilu-2.0a12 has to be replaced with the actual
ILUHOME/lib in both cases. For the Setup file, needs to be done prior
to the installation.

Regards,
Martin
```

The file '*ILUHOME*`/lib/Setup`' exists in this distribution, so it should be possible to 'freeze' a Python image containing ILU with the commands

```
% python -O freeze/freeze.py -o outdir -e ${ILUHOME}/lib myscript.py
```

assuming that '`myscript.py`' contains your Python program, and that '*ILUHOME*`/lib`' is on your `PYTHONPATH` environment variable.

4.5 Python/ILU API Reference

4.5.1 Identifiers Exported From Module `ilu`

AutoImport ([*path*=() [*verbose*=0]]) Function

If called, enables the auto-loading of '`.isl`' and, if OMG IDL support is configured into ILU, '`.idl`', files that are on the user's `ILUPATH` environment variable. The `python-stubber` program is run to generate the Python surrogate stubs for the interface description into a temporary directory, and those stubs are loaded into the current program. The stubber is re-run every time the interface is imported. If an error occurs while producing the Python stubs, an exception is raised and the import process stops. The *path* parameter has no effect; the *verbose* parameter will cause various messages to be written to the standard output during the process of importing, if set to a non-false value.

CallerIdentity () Function

Returns the passport containing identities of the caller. This routine is only valid inside the code of a true method.

CORBAMapping Constant

A value which evaluates to Python boolean True if the CORBA mapping for Python has been selected, and False if the 'classic' ILU mapping has been selected.

CreateAlarm () Function

Creates an object of type `ilu_Alarm`.

CreateLoopHandle () Function

Creates and returns an instance of a ''loop handle'' object, which can be passed to `ilu.RunMainLoop` and `ilu.ExitMainLoop().`

CreatePassport () Function

Creates and returns an empty instance of a `ilu_Passport` object. The `ilu_Passport` object is used to provide a sense of identity in the ILU system. It can hold any number of different identities, each of which is represented with an appropriate data structure that varies from identity type to identity type.

The `ilu_Passport` object type has the following methods:

- `lookupIdentity (`*IDENTITY-TYPE-NAME*`)` - returns the data structure for the specified identity, if the passport contains one; Returns `None` otherwise. Raises `ilu.IluGeneralError` if the named identity type does not exist. The identity type `'ConnectionIdentity'` is always supported; the identity type `'SunRPCAuthUnixIdentity'` will be supported if support for the `sunrpc` protocol has been configured into ILU.
- `addSunRPCAuthUnix (`*HOSTNAME*`,` *UID*`,` *GID*`,` *TUPLE-OF-GROUPS*`)` - adds an identity of type `'SunRPCAuthUnixIdentity'` to the passport with the specified *HOSTNAME*, *UID*, *GID*, and *TUPLE-OF-GROUPS*. See Appendix A of RFC 1831 at `http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1831.txt` for

details on the meaning of these parameters. This method will only be available if support forthe `sunrpc` protocol has been configured into ILU.

CreatePipeline () Function

Creates and returns an empty instance of a `ilu_Pipeline` object. The `ilu_Pipeline` object is used to allow multiple requests to be outstanding on non-concurrent protocol streams. The `ilu_Pipeline` object has no methods.

CreateServer ([*serverID* [*transport* [*protocol* [*objtable*]]]]) Function

Used to create an `ilu_Server` object with the specified serverID, transport, and protocol. If *serverID* is unspecified or `None`, an identifier will be invented automatically. If transport or protocol are unspecified or `None`, they will default to `('sunrpcrm', 'tcp_0_0')` and `'sunrpc'`, respectively. (Other combinations that would work are transport of `('tcp_0_0')` and protocol of `'iiop_1_0_1'`, transport of `('sunrpcrm', 'tcp_0_0')` and protocol of `'courier'`, and transport of `('tcp_0_0')` and protocol of `'http'`, depending on the configuration of your ILU system.) The first time `CreateServer` is called, the server so created becomes the default server. If there is no default server when one is required, one will be created using default parameters and a message will be issued on stderr. The *objtable* argument allows specification of a callback function for creating true instances on demand. The callback function should take one argument, a string, which is the instance handle of the instance to be created, and return a true instance.

An `ilu_Server` object has the following methods:

- `id ()` - returns the string identifier of that server.
- `addPort (`*TRANSPORT*`,` *PROTOCOL*`)` - adds a port with the specified *TRANSPORT* and *PROTOCOL* (described above) to the server instance.
- `createSerializer ()` - creates and returns a new serialization context object.
- `setRelocator (`*RELOCATOR-FUNCTION*`)` - sets the relocation procedure of the server to be *RELOCATOR-FUNCTION. RELOCATOR-FUNCTION* should be a function with no arguments, that returns either `None`, or a tuple with two elements. If it returns the two-element tuple, the first element should be a string specifying pinfo as in the arguments to `CreateServer`, and the second argument should be a tuple of tinfo, again as in the arguments to `CreateServer`.

DefaultServer () Function

Returns the default server.

DictionaryPassing Constant

A value which evaluates to Python boolean True if the configuration option *--enable-python-dictionaries* has been selected, and False otherwise. That is, this is True if certain sequence types are mapped to Python dictionaries.

DoSoon (*FUNCTION, ARGS-TUPLE, STRING-DESCRIPTION*) Function

Causes the function *FUNCTION* to be run with args *ARGS-TUPLE* to be run at some point in the future, when the system finds it to be convenient. In the threaded world, a new thread is forked to run the function; in the non-threaded world, the function is executed at some point by the event loop as a background task.

ExitMainLoop (*loophandle*) Function

Exits the ILU main loop, assuming it is running. The *loophandle* is created by a call to `ilu.CreateLoopHandle()`, and must have been previously used as an argument to a call to `ilu.RunMainLoop()`.

FALSE Constant

A value which evaluates to Python boolean False.

FineTime (*seconds*) Function

Converts its `int` or `float` argument *seconds* in units of seconds to type `ilu_FineTime`. Objects of this type can be compared, added, subtracted, and converted to `int` or `float`. The main use of objects of this type is in setting alarms.

FormSBH (*sid, ih, type, pinfo, tinfo_vec*) Function

Forms a valid ILU string binding handle from the arguments and returns it. The *sid* and *ih* arguments are strings containing the server ID and instance handle for the desired instance. The *type* argument should be the Python class for the most specific object type of the desired object. The *pinfo* argument is a tuple containing the protocol information describing the object implementation's preferred communication protocol. The *tinfo_vec* argument is a tuple of tuples, specifying the transport stack needed to connect to the implementation. Each sub-tuple in the *tinfo_vec* is a tuple describing a particular transport layer.

For instance, to create a string binding handle for an instance of type `Foo.Bar`, with server id `"some-server-id"` and instance handle `"some-instance-handle"`, exported via `Sun RPC`, version 2, with program number 1000007, version 3, via `TCP/IP` from host `"foobar.somewhere.com"`, port 3456, we'd say

```
sbh = ilu.FormSBH('some-server-id', 'some-instance-handle', Foo.Bar,
('sunrpc_2', 1000007, 3), (('sunrpcrm',), ('tcp', 'foobar.somewhere.com',
3456)))
```

Note the comma used after `'sunrpcrm'` to create a true tuple; note also that use of this procedure requires some specialized knowledge, such as knowing that use of `Sun RPC` also requires use of the `Sun RPC` record-marking transport layer when used over `TCP/IP`.

FineTimeRate Constant

The precision of type `ilu_FineTime` in seconds is the reciprocal of this constant.

FineTime_Now () Function

Returns the current time as an `ilu_FineTime` object.

FormSBH (*objectID*, *contactInfo*) Function

Returns the string binding handle corresponding to the object id *objectID* and contact info *contactInfo*. This is the inverse of `ParseSBH`.

GetFDBudget () Function

Returns the current setting of the file descriptor budget.

GetPassport () Function

Returns the current passport for this thread. See also `CreatePassport()` and `SetPassport()`.

GetPipeline () Function

Returns the current pipeline context for this thread. See also `CreatePipeline()` and `SetPipeline()`.

GetSerializer () Function

Returns the current serialization context for this thread. See also the `createSerializer()` method on the `ilu_Server` class, and the `SetSerializer()` function.

IluGeneralError Exception

An exception that may be returned from the ILU runtime. This exception is used to return all 'standard' exceptions, with a string value to indicate the specific type of standard exception that occurred.

IluProtocolError Exception

An exception that may be returned from the ILU runtime. This exception is raised for all on-the-wire exceptions, with a value that indicates which kind of protocol exception occurred.

IluUnimplementedMethodError Exception

An exception that may be returned from the ILU runtime. Raised when an unimplemented method is called, typically on a true instance.

IluUnknownTypeIDError Exception

An exception that may be raised from the ILU runtime. It indicates that the associated type ID value is unknown in this address space.

IOROfObject (*obj*) Function

If the `IIOP` protocol has been configured in, returns the string IOR of the object, as specified in the CORBA 2 IIOP specification. If the `IIOP` protocol has not been configured in, throws an error.

LongReal (*v*) Function

Converts its `int`, `float`, or sixteen-integer `list` or `tuple` argument to type `ilu_LongReal`. In case of a list or tuple, the elements encode the bytes of the IEEE long real value, from most significant to least.

LookupObject (*sid, ih, cl*) Function

Returns the object with object server ID *sid*, object instance handle *ih*, and Python class `cl`, assuming it was previously published using the simple binding service. If the lookup fails, `None` is returned.

ObjectOfSBH (*cl, sbh*) Function

Returns the object corresponding to the Python class *cl* and string binding handle *sbh*.

ParseSBH (*sbh*) Function

Returns the pair (object id, contact info) corresponding to the string binding handle *sbh*.

RegisterCustomSurrogate (*class*) Function

Registers *class* as the object type to create when receiving a surrogate of the type indicated by the `_IluClass` field of *class*. *class* must be a subtype of the default surrogate type for this ILU type. This allows custom surrogates, with implications for caching and other object-type-specific functions.

RegisterInputHandler (*file, handler_fn*) Function

Sets up an association between the *file* (which must be a file object opened for reading), and the *handler_fn* (which must be a callable function with no arguments) so that *handler_fn* is called whenever input is available on *file*. This is useful for implementing a server that also responds to commands typed to its standard input, for example. Passing a value of `None` for the *handler_fn* removes the association. This procedure should only be used in non-threaded applications. In threaded applications, you should fork a thread to handle this, instead.

RegisterOutputHandler (*file, handler_fn*) Function

Sets up an association between the *file* (which must be a file object opened for writing), and the *handler_fn* (which must be a callable function with no arguments) so that *handler_fn* is called whenever input is available on *file*. Passing a value of `None` for the *handler_fn* removes the association.

RunMainLoop (*loophandle*) Function

Runs the ILU main loop. The argument *loophandle* is a ''handle'' on that loop invocation, created by a call to `ilu.CreateLoopHandle()`. This function can be used with either threaded or non-threaded use of ILU/Python. In the threaded use, it simply runs `sleep.sleep` in the calling thread.

SetCalloutExceptionHandler (*handler-fn*) Function

This function can be used to define a function *handler-fn* which is called when an internal Python exception is signalled in code called from the ILU C code. The handler function receives four arguments: a string indicating where in the ILU runtime the exception was encountered, the exception type, the exception value, and a traceback object. This function is typically used to note the exception to a file or stderr; see the example usage in '*ILUSRC*`/runtime/python/iluRt.py`'. If a parameter of `None` is passed to `SetCalloutExceptionHandler`, it cancels any handler function in use, and a default built-in one is used.

SetDebugLevel (*flags-or-switches*) Function

Sets the ILU kernel debugging flags according to its `int` argument, if an int is specified, or via the colon-separated list of debug switches, if a string is specified. See the Debugging section of the ILU Manual for more information on these switches.

SetFDBudget (*desiredbudget*) Function

Attempts to set the file descriptor budget. Returns what the new budget actually is (may be different than requested).

SetMainLoop (*DoEvent, RegisterInput, CancelInput, RegisterOutput, CancelOutput, CreateAlarm, SetAlarm, CancelAlarm*) Function

The purpose of this function is to be able to use a foreign main loop (such as for a user interface toolkit) with an ILU server. The details will not be described here. Look at the runtime module `ilu_tk` for an example of its use. This function should only be used with non-threaded use of ILU/Python.

SetPassport (*passport*) Function

Sets the current passport identity for this thread, and returns the passport active before this call. Either of these can be `None`. Also see the function `CreatePassport`, and the function `GetPassport`.

SetPipeline (*pipeline*) Function

Sets the current pipelining context for this thread, and returns the context active before this call. Either of these can be `None`. Also see the function `CreatePipeline`, and the function `GetPipeline`.

SetSerializer (*serializer*) Function

Sets the current serialization context for this thread, and returns the context active before this call. Either of these can be `None`. Also see the `createSerializer` method on the class `ilu_Server`, and the function `GetSerializer`.

TCPDefaultBufferSize (*size*) Function

Sets the default buffer sized used for TCP/IP transport buffers to *size*. Returns the previous default buffer size. Raises `IluGeneralError` if support for the TCP/IP transport is not configured into ILU.

TCPStatistics (**reset*) Function

Returns a dictionary containing the current TCP/IP statistics for this process. If a True value is specified for the optional argument, the statistics counters are reset. If TCP/IP support is not configured into ILU, this routine will raise the exception `IluGeneralError`.

ThreadedOperation () Function

Enables thread use in both the ILU kernel and the ILU/Python runtime. This routine should be called before any other ILU calls are made.

TRUE Constant

A value which evaluates to Python boolean True.

TypeID (*cl*) Function

Returns the ILU unique type identifier corresponding to the Python class *cl*.

TypeName (*cl*) Function

Returns the ILU type name corresponding to the Python class *cl*.

Version Constant

The ILU version string.

4.5.2 Identifiers Exported from the `CORBA` Module

InitialReferences Variable

A dictionary with string keys, and values of type `CORBA.Object`. It is used to resolve strings passed as parameters to `CORBA.ORB.resolve_initial_references()`. The following names are supported automatically by Python runtime:

- **NameService**
  If the environment variable `ILU_COS_NAMING_IOR` is bound to a string IOR for a OMG IDL `CosNaming::NamingContext` object instance, the Python runtime will attempt to create a surrogate for that instance locally, ping it, and if successful will bind it to the string `"NameService"`.

InvalidName Exception

Raised when an invalid name is passed to `CORBA.ORB.resolve_initial_references()`. Has the associated bad name as its value.

Object (ilu.IluObjSurr) Class

A type which all object types defined in OMG IDL, or inheriting from `ilu.CORBA-Object` in ISL, participate in. It supports the following methods:

- `_is_a(`*type_uid*`)` - returns `True` if the object is of the specified type, `False` otherwise; raises `ilu.IluUnknownTypeIDError` if the *type_uid* is unknown in this address space;
- `_is_nil()` - returns `False`; raises `TypeError` if called via `CORBA.Object._is_nil()` on a non-Python-object type;
- `_non_existent()` - returns the logical inverse of the result of calling `ilu.PingObject()` on the object;
- `_is_equivalent(`*other*`)` - returns the result of comparing *self* and *other* with the Python == operator;
- `_duplicate()` - does nothing, returns *self*;
- `_release()` - does nothing, returns nothing;
- `_hash(`*max_value*`)` - returns `(hash(`*self*`) % (`*max_value* `+ 1));`
- `_get_implementation()` - raises `IluUnimplementedMethodError`;
- `_get_interface()` - raises `IluUnimplementedMethodError`;

The `CORBA.Object` class is actually implemented in `iluRt.CORBA_Object`, so all classes which inherit from `ilu.CORBA_Object` will have access to these methods.

ORB Class

The general class for manipulating the object request broker. There is typically only one instance of this class per address space. It is retrieved with the function `CORBA.ORB_init()`; it supports the following methods:

- `object_to_string(`*instance*`)` - returns a string which can be used in a subsequent `string_to_object()` call;
- `string_to_object(`*string*`)` - if the specified *string* is well formed and specifies an object, the object is created locally and a reference is returned; the reference may be to a true instance if the string names a true instance; if the string is poorly formed, the Python exception `ilu.IluGeneralError` is raised. This method does not test for the existence of the instance.
- `resolve_initial_references(`*string*`)` - If the *string* argument is bound in the dictionary `CORBA.InitialReferences`, the value is returned. Otherwise, the exception `CORBA.InvalidName` is raised. See the documentation of `CORBA.InitialReferences` for a listing of the names that are bound automatically, if any.

ORB_init (*argv*=(), *orb_id*='*ilu*') Function

Returns an instance of `CORBA.ORB` with the specified *orb_id* (currently only the ORB ID `'ilu'` is supported). The arguments which may be passed in via *argv* are ignored.

4.5.3 Methods and Attributes of ILU Objects

- `IluObjectID()` returns the object ID of the object.
- `IluPublish()` publishes the object using the simple binding service.
- `IluSBH()` returns the object's string binding handle.
- `IluTypeID()` returns the unique type identifier of the object's ILU type.
- `IluTypeName()` returns the type name of the object's ILU type.
- `IluWithdraw()` undoes the effect of IluPublish().

Special attributes of ILU true objects: One or more of the following attributes may be set in a true (implementation) object of an ISL object type to control certain aspects of that object.

- `IluInstHandle`, a string instance variable, gives the object's instance handle. If not present, an instance handle is invented automatically.
- `IluServer`, a variable of type `ilu_Server`, determines the object's server. This can be an instance or a class variable. If not present, the default server is used.

5 Using ILU with CORBA 2.0 C++

5.1 Warning

The CORBA 2.0 C++ stubber is limited in functionality. The stubber is only currently working for Visual C++ (sans nested modules and pickles/anys support). This is very new software - it has received very limited testing, and the stubber receives little support from the current team.

5.2 Introduction

This chapter describes the use of ILU with C++ in a manner compliant with the CORBA 2.0 C++ language mapping specification. (see `http://www.omg.org/corba/corbiiop.htm`)The use of ILU's original C++ support is deprecated.

Any function or type which is not part of the CORBA 2.0 specification has the prefix 'ilu'. It should be understood that use of 'ilu' prefixed functionality is not portable to other (non-ILU) CORBA implementations.

Some arguments or return values of functions (e.g. char*) have storage management requirements. Basically this revolves around whether the caller retains or gets ownership of the parameter and is therefore responsible for eventually releasing it, or if ILU takes or retains ownership, where it will be released at ILU's discretion. Any function parameter that becomes owned by ILU is marked with the comment /* ILUowned */ Any return value (or 'out' parameter) that remains under the ownership of ILU is similarly marked. Anything not so marked is not ILU's responsibility.

Note that ILU support for C++ does rely on having argument prototypes, all C++ library functions, and the capabilities of the C++ pre-processor.

5.3 Mapping ILU ISL to C++

The CORBA 2.0 C++ chapters 15 though 18 describes the mapping of OMG IDL to C++. For those elements of ISL for which there is a direct counterpart in IDL, the ISL component is mapped just as the IDL component is. Those ISL concepts with no IDL counterpart (marked with a '–' in the table below) have a mapping separately described in a following section.

5.3.1 ISL to IDL Correspondences

```
ISL                     IDL
-------------------------------------
INTEGER                 long
SHORT INTEGER           short
LONG INTEGER            -
CARDINAL                unsigned long
```

```
SHORT CARDINAL         unsigned short
LONG CARDINAL          -
BYTE                   octet
BOOLEAN                boolean
REAL                   double
SHORT REAL             float
LONG REAL              -
CHARACTER              -
SHORT CHARACTER        char
PICKLE                 Any
ARRAY                  array
SEQUENCE               sequence
RECORD                 struct
UNION                  union
OPTIONAL               -
ENUMERATION            enum
OBJECT                 object
CString                string
SEQUENCE OF CHARACTER  -
EXCEPTION              -, exception
INTERFACE              module
```

5.3.2 ISL Specific Mappings

The following table describes the mappings for ISL types that have no IDL counterparts. The C++ column gives the mapping modulo indirection and/or 'const' qualification dictated by parameter directionality (i.e., IN vs OUT vs INOUT vs return values).

```
ISL                    C++
---------------------------------------------------------------
LONG INTEGER           iluLongInteger
LONG CARDINAL          iluLongCardinal
LONG REAL              iluLongReal
CHARACTER              iluCharacter
OPTIONAL X             X* for operation parameters;
                       'managed X*' for embedded types.
SEQUENCE OF CHARACTER  iluCharacter*
EXCEPTION              Any ISL exception that is
                       not a RECORD, maps to a subclass of
                       CORBA::UserException, that has a _value()
                       member function which returns a value of
                       the type associated with the exception.
```

5.3.3 Correspondence between C++ Types and Kernel Types

To provide a consistent naming scheme, in many cases, a type defined in the ILU kernel has been typedeffed to appear in C++ as the corresponding type name without the intervening underscore, and with the following letter capitalized, e.g. typedef ilu_cardinal iluCardinal;

5.3.4 C++ Classes Produced for an Object

The mapping for an ISL Object 'A' produces 3 C++ classes:

1. The A C++ class, which has pure virtual member function declarations for each of A's methods. For each ISL Object supertype Si of ISL Object A, the C++ class A 'public virtual' inherits from the Si C++ class. If the ISL Object A has no supertypes, the A class 'public virtual' inherits from the iluObject class. [NOTE that an object described in IDL will implicity inherit from CORBA::Object (which in turn inherits from iluObject). The idl2isl translator automatically adds ilu.CORBA-Object as a SUPERTYPE.] This basically creates a C++ class hierarchy that is isomorphic to the ISL object type hierarchy, with each method being declared pure virtual. We refer to this hierarchy as the 'abstract object hierarchy'.
2. The A_surrogate C++ class, which has virtual member function declarations for each of ISL Object A's non inherited methods. These member functions transfer requests to the true object. For each ISL Object supertype Si of ISL Object A, the C++ class A_surrogate 'public virtual' inherits from the Si_surrogate C++ class, and then 'public virtual' inherits from the C++ class A. (If the ISL Object A has no supertypes, the A_surrogate class 'public virtual' inherits only from the C++ class A.) This basically creates a C++ 'surrogate object hierarchy' that is isomorphic to the abstract object type hierarchy, with the addition that each X_surrogate class also inherits from its counterpart in the 'abstract object hierarchy'.
3. An A_var C++ class as prescribed by the CORBA C++ mapping.

This mapping allows servers to be developed that do not contain surrogate stub code (if they don't need it), and also prevents the situation where a server method override is forgotten, resulting in surrogate stub code being called as it it were 'true' code.

For each ISL Object 'A' for which a true side implementation is to be developed, the true side implementer should define a class A_impl that inherits virtually from the C++ class A, and implements the actual true methods as member functions (in whatever manner is appropriate). The implementer is free to use delegation, implementation inheritance, whatever - the only restriction is that if a class B_impl inherits implementation from a class X, and class X inherits from a class in the abstract object hierarchy, (e.g. when X is A_impl), then X's inheritance from the abstract object hierarchy must be 'public virtual'.

Each produced C++ class e.g. A, will have a constructor

A (char* *pc_instance_handle*, ilu_Server& *r_an_ilu_server* = iluServer::GetDefaultServer(), **ILUCPP_BOOL** *b_within_object_table* **= ILUCPP_FALSE) : iluObject (** *A::m_ILUClassRecord*, *pc_instance_handle*, *r_an_ilu_server*, *b_within_object_table***).** C++

5.3.5 Misc. Mapping Details

5.3.5.1 Unions

In IDL, Union arms all have names, in ISL, the names may not be specified. If a name isn't specified for an arm, the name the stubber produces is the arm's typename, prefixed with '_', and suffixed with '_arm'.

For example,

```
TYPE someuniontype = short cardinal UNION
        bar = 0, 1 END,
        integer = DEFAULT
END;
```

would produce the C++ names _bar_arm, and _integer_arm to reference the bar and integer arms.

5.3.5.2 Optionals

For procedure parameters, an ISL `OPTIONAL` type maps either to the same C++ type as its base type, if that base type is represented with an C++ pointer type, or to a pointer to that base type, if it is not represented with a C++ pointer type.

Additionally, all `OPTIONAL` types *T* have an associated C++ *T*_var.

For non-parameters (i.e., `RECORD` members, `ARRAY` and `SEQUENCE` elements) an ISL `OPTIONAL` type maps to a 'managed pointer', analagous to the mapping for non-parameter ISL `OBJECT` (CORBA `interface`) and ISL `SEQUENCE OF SHORT CHARACTER` (CORBA `string`). (This 'managed pointer' behaves similarly to *T*_var; however, CORBA does not allow compliant applications to use 'managed pointer' types directly, as the actual type is implementation-specific)

The following ISL and C++ code fragments illustrate:

```
     ISL

     TYPE SomeType   = ...
     TYPE MyOptional = OPTIONAL SomeType;

 TYPE MyRec = RECORD
     member: MyOptional
 END;

 TYPE MyArray = ARRAY OF 10 MyOptional;
```

```
    member: MyOptional
END;

    C++

...

    MyRec          myRec;
MyArray        myArray;
SomeType       st;
SomeType*      stPtr;
MyOptional     myOptional;  // MyOptional is equivalent to SomeType*
MyOptional_var stVar;

...

myRec.member = myArray[0];         // free old myRec.member, deep copy
myArray[1]   = stVar;              // free old myArray[1], deep copy
myRec.member = stPtr;              // free old myRec.member, assume ownership
myRec.member = myOptional;         // free old myRec.member, assume ownership

myRec.member = &st;                // ILLEGAL - can't free &st
myRec.member = new SomeType(st);   // OK; free old myRec.member, assume ownership

stPtr = myArray[2];                // Simple pointer assignment, no copy
stVar = myRec.member;              // free old stVar, deep copy
```

The lifetime of `recA.member` and each `myArray[n]` are tied to recA and myArray, respectively; when an optional-containing variable goes out of scope or is destroyed, its optional members/elements are freed. Thus, the assignment `myRec.member = &st` in the example above is illegal and can lead to calamitous results when an attempt is made to free &st.

5.3.5.3 Exceptions

Any ISL exception that is not a RECORD, maps to a subclass of CORBA::UserException, that has a _value() member function which returns a value of the type associated with the exception.

5.3.5.4 ISL Asynchronous Methods

In IDL, methods may be `ASYCHRONOUS`. Asynchronous methods cannot have return values or raise exceptions. Hence, they result in a C++ member function declared to return void.

5.3.5.5 ISL Functional Methods

In IDL, methods may be `FUNCTIONAL`. In the C++ mapping, `FUNCTIONAL` is ignored. The ability to create custom C++ surrogates allows the implementation to decide what and how caching may be implemented on any method, as well as perform any other sorts of message 'filtering'.

5.3.5.6 ISL Collectible Objects

An ISL object being declared `COLLECTIBLE` has not effect on the mapping per-se. It will however cause ILU to adjust an object's reference count based on interest or dis-interest from clients.

5.3.5.7 Inheritence from CORBA::Object

If an ISL Object A has no supertypes, the A class 'public virtual' inherits from the iluObject class. An object described in IDL will implicity inherit from CORBA::Object (which in turn inherits from iluObject). (The idl2isl translator automatically adds ilu.CORBA-Object as a SUPERTYPE.) So, if you define an object in ISL, and do not explicitly declare ilu.CORBA-Object as a SUPERTYPE, you will not have the member functions of CORBA::Object available since you do not inherit from it.

5.3.5.8 Portability and Mapping Variations

The CORBA 2.0 C++ mapping allows for variations in the mapping depending on the C++ compilers support for Name Spaces, Exception Handling, and Run-Time Type information.

The ILU CORBA 2.0 C++ mapping implementation assumes that the C++ compiler supports exceptions. We also assume that the compiler supports RTTI should someone want to do narrowing within the exception hierarchy. [Given that ILU does not provide a Dynamic Invocation Interface, there's no real need to narrow exceptions anyway.]

During the configuration phase of ILU installation (or for Windows, per the definitions in '*ILUSRC*`/runtime/kernel/iluwin.h`') a determination is made as to whether or not to use namespaces, nested classes or underscores for IDL modules, based on the C++ compiler in use. [This can be overridden with the configuration option --with-cplusplus-mapping= switch to config.] This results in a C++ runtime that is constructed with one of these approaches in mind. Should a user decide to use a different mapping, this can be done by setting appropriate switches to the stubber, but you must bear in mind that the C++ runtime needs to be in correspondence, since the selection radically changes the names seen by a linker. Based on our knowledge (as of the date of this writing), of the degree of support/bugs for namespaces and nested classes, the following describes the IDL module mapping based on compiler:

C++ Compiler Module Mapping

```
Compiler                 Mapping
---------------------------------------
Microsoft Visual C++     underscores
```

```
SunPro                    nested classes
Gnu                       nested classes
```

To portably reference things in a interface when the stubber is directed to produce portable code, the stubber generates two macros, one for referring to things from outside the namespace (e.g. when defining a member function), and another for inside the namespace (e.g. inside a class declaration). The names of these macros are the interface name, and the interface name with a '_' (underscore) suffix. These macros expand to some other macros that are defined in the cppportability.hpp file.

To refer to things in the CORBA interface, one uses the CORBA macro. For example, CORBA(Boolean) b_my_result = ILUCPP_TRUE;

Because of possible variations in compiler support for Booleans, CORBA(Boolean) is defined as ILUCPP_BOOL, where ILUCPP_BOOL is defined as either an int (with ILUCPP_TRUE and ILUCPP_FALSE #defined as 1 and 0), or as a bool (with ILUCPP_TRUE and ILUCPP_FALSE #defined as true and false).

5.4 Concepts

5.4.1 Servers and Ports

In ILU there is a concept of an 'server object'. In the kernel this is the ilu_Server, which in the C++ runtime is encapsulated asn an `iluServer` object. This 'server' effectively forms a 'scope' in which true objects reside. This is why for example, and object lookup requires both the 'server' ID, and the object's instance handle - both are needed to uniquely denote an object.

Now a server has some number of 'ports'. A port is basically a means of communicating with the objects inside a server, using a particular combination of protocol and transport. For example, when create an `iluServer`, the constructor for `iluServer` automatically adds a port for the communication protocol and transport specified as constructor arguments. We can call `iluServer::iluAddPort` to have additional ports added. For example we may want to be able to communicate with the objects using `sunrpc` over `tcp/ip`, as well as `http` over `tcp/ip`. The iluServer has a notion of a default port. This is initially one as specified during construction, but this can be changed if when calling `iluServer::iluAddPort` we specify that this should become the default port. The default port is the one used when we ask for contact information for an object - that is, if we get the string binding handle for an object, the contact information in that string will reflect the default server port.

5.4.2 Object Tables

True objects may either be created ahead of time, or on an 'as needed' basis, i.e. when a call comes in involving them. The 'as needed' situation is made possible by 'object tables'. An `iluServer` may have associated with it (at construction time) an `iluObjectTable` object. When a call comes involving an object in that `iluServer` that the ILU doesn't already know about, the `iluObjectTable` object's `iluObjectOfInstanceHandle` gets called. It is the job of this function to create and return a new object with that instance handle. How it does this is specific to your application - it may read object state off a disk for example. In any event, one thing this function must do is ensure that when it calls the true objects constructor, that it sets the constructor's b_within_object_table argument to true. (Otherwise, internal locking constraints will be violated). While in the `iluObjectOfInstanceHandle` function, the associated `iluServer`'s lock is held, and if the resulting object is expected to be of a COLLECTIBLE type, the global kernel mutex `"gcmu"` is also held. The fact that these locks are held somewhat restricts what an application can do inside this mapping procedure.

5.4.3 Threading

The ILU C++ support may be initialized to run in either threaded or non-threaded mode. In non threaded mode, a call to `iluServer::iluRun` member function results in a call to ILU's 'mainloop'. The mainloop basically sits waiting for an incoming request. When one comes in, the request is invoked. If the implementation of the invoked method makes a call to some other remote object, the mainloop is recursively entered while awaiting a reply. This allows additional requeste to come in and get serviced, preventing deadlock.

When intialized to run in threaded mode, ILU will run one thread for each incoming connection. Note that there may be multiple connections for a particular port (either from different clients, or from the same client who needed another connection because all the ones it had so far were busy at the time). In the case of a non-concurrent protocol (sunrpc, http, courier), the connection thread receives an incoming request, processes it itself, and then waits for the next request. In the case of a concurrent protocol (csunrpc, iiop), the connnection thread receives an incoming request, spawns a worker thread to carry out the request, and immediately goes back to waiting for more incoming requests.

The ILU C++ provides no special concurrency control for methods in your objects (to do so would be presumptive on our part). The method implementor must put appropriate locking in place if it is possible that multiple threads (or recursive mainloop invocations) might be running 'in' an object simultaneously.

5.4.4 Custom Surrogates

A surrogate is an object that is used to represent a remote object. When a method is invoked on a surrogate, the methods implementation in the surrogate transfers the call to the true object, and returns the result of this call, thus providing location transparency. There are times however when it is useful to have the surrogate's method implementation do more than just forward the call to the true object. An application may want a surrogate method implementation that caches the results of calls (potentially reducing network overhead), perform transformations on arguments, output diagnostic information, or whatever.

To facilitate this, the ILU C++ support allows an implementation to supply a function that is called when a surrogate for a particular object type is needed. The function `iluCppRuntime::iluSetSurrogateCreator` tells the C++ runtime what function to call when a surrogate for an object of the specified class is needed.

This allows an implementation to subclass off a surrogate class, and write a new surrogate creation function that creates an instance of this new subclass. Call `iluCppRuntime::iluSetSurrogateCreator` after you've performed initialization, but before you do any operations which might create a surrogate of the specified class. It basically overwrites the default surrogate creation function set up by the surrogate stubs. It returns the old surrogate creator function, or `NULL` if was previously no surrogate creator for that class.

A surrogate creator function should at the minimum create an instance of a surrogate, call the instances member function iluAssociateKernelObject passing the `iluKernelObject`, and then return a pointer to the new instance.

5.4.5 String Binding Handle Manipulation

A String Binding Handle is a textual representation of an object reference. It contains the object's server id, instance id, information about how to contact the object, as well as other information. ILU C++ provides the functions `iluCppRuntime::iluFormSBH`, `iluCppRuntime::iluFormSBHUsingContactInfo`, and `iluCppRuntime::iluParseSBH` for constructing and parsing string binding handles. An object may be obtained from a string binding handle using `iluObject::iluStringToObject` and the string bindign handle of an object may be obtained by calling the `iluObjectToString` member function.

5.4.6 Simple Binding

When creating a service, there needs to be some way for clients to find out about the service. ILU C++ provides a simple mechanism to achieve this. Objects may be published, looked up, and their publications withdrawn using the appropriate member functions (`iluPublish`, `iluLookup`, `iluWithdraw`).

5.4.7 Object Activation

An true object is initially 'Active', which means that its ISL (or IDL as the case may be) defined methods may be invoked on it from outside its process (or from another language within that same process). An object may be made unavailable to outside calls, i.e. marked 'inactive' by calling its iluDeactivate member function. It may may be reactivated by calling its iluActivate member function.

An object is initially available from the outside until it is deactivated Objects that are involved in a call (i.e. sent or received as arguments, or the object the method is being invoked on) need to be protected from deletion for the duration of that involvement (for example, you don't want some thread deleting a true object when it's currently the target of a method call). The C++ runtime keeps track of what objects are involved in a call, and will attempt to prevent them from being deleted until the call is completed.

The application programmer needs to assist in this by calling, in the most specific destructor, iluDeactivate (inherited virtually from `iluObject`). iluDeactivate blocks any further incoming calls involving the object, and wait for any ongoing calls using the object to complete. Next the destructor should perform any object specific cleanup. Finally, the destructor in `iluObject` will break the association between the kernel object and this object, allowing the kernel object to be potentially freed.

5.4.8 Security

A client may set the Passport to be used on outgoing calls by creating and setting up an `iluPassport`, and then passing the passport in a call to iluPassport::iluSetPassport. This sets the passport to be used in the thread that made the call - i.e. `iluPassport` are on a per thread basis. Note that before your thread exits, you should either call iluSetPassport(`NULL`), or delete the `iluPassport` in use (assuming it's only in use for a single thread). The `iluPassport` (if any) currently setup for a thread can be retrieved by calling iluPassport::iluGetPassport.

A Server may obtain the `iluPassport` of the caller (if any) of a method by using the iluPassport::iluGetCallerPassport() function.

A `iluServer` may be constructed to use a particular identity by specifying a `iluPassport` as a constructor argument. This identity is used to identify the principal offering the service.

5.4.9 Static Initialization

The C++ Runtime normally relies on the static initializers in the files that the stubber generates to place initialization functions onto internal lists so that they will be invoked when the application calls iluCppRuntime::Initialize. However, it is not guaranteed by the ANSI C++ that static initializers are called upon the loading of a compilation unit. We have only had a report of one compiler that did not run the static initializers at load time (in fact, it was reported that it did not run them ever! - bug!?). We have observed static initialization at load time in Visual C++, SunPro and GNU compilers. In the event that you end up using a compiler that does not call the static initializers at load time, you can use the stubber defined initialization macros that are generated in the common header file for each interface.

(It should be pointed out that the CORBA 2.0 C++ Runtime does not suffer from the static initializer issues that plagued ILU's original C++ support. No ILU calls are actually made until iluCppRuntime::iluInitialize is called, allowing one to set up different mainloops, etc.)

5.5 Building an Application

5.5.1 Running the Stubber

To generate CORBA 2.0 C++ stubs from an ISL file, use the program cpp2-stubber. The stubber has the following usage:

```
    Usage: cpp2-stubber [arguments] Islfile [ISLFILE ...]
[arguments] can be any of the following:
[-nu|-underscores]
[-np|-portable]
[-nn|-nested]
[-ns|-namespaces]
```

The switches can be used to direct the stubber to produce code that using a specific module mapping, or portable module mapping. The default is whatever is found appropriate during the configuration phase of ILU installation (see "Portability and Mapping Variations"). Note that generating a specific (non-portable) mapping that does not match the platform default, is likely to cause errors at link time due to naming differences.

5.5.2 Stubber Generated Files

For an interface `Foo` the stubber generates:

'`Foo-cpp.hpp`' which contains the classes for the abstract object hierarchy, as well as any other declarations needed by both client and server.

'`Foo-cpp.cpp`', which contains any definitions needed by both client and server

'`Foo-cppsurrogate.hpp`' which contains the classes for the surrogate object hierarchy, as well as any other declarations needed just by a client. This file #includes Foo-cpp.hpp

'`Foo-cppsurrogate.cpp`', which contains any definitions needed just by a client. This file #includes Foo-cppsurrogate.hpp

'`Foo-cpptrue.hpp`' which contains any declarations needed just by a server. This file #includes Foo-cpp.hpp

'`Foo-cpptrue.cpp`', which contains any definitions needed just by a server. This file #includes Foo-cpptrue.hpp. All header files use the usual #ifdef method to prevent multiple inclusions.

A client only will #include Foo-cppsurrogate.hpp, and link with '`Foo-cpp.o`' and '`Foo-cppsurrogate.o`'

A server only will #include Foo-cpptrue.hpp, and link with '`Foo-cpp.o`' and '`Foo-cpptrue.o`'

A client and server will #include Foo-cpptrue.hpp, #include Foo-cppsurrogate.hpp, and and link with '`Foo.o`', '`Foo-cpptrue.o`', and '`Foo-cppsurrogate.o`'

5.5.3 Server Basics

The basic steps in creating a simple server application are as follows (assuming we have a ISL file called '`foo.isl`', describing an interface 'foo' with an object type 'bar'):

1. Run the C++ stubber on '`foo.isl`', e.g. cpp2-stubber foo.isl
2. In your implementation file, e.g. '`servermain.cpp`', include the true side header file, e.g. #include "foo-cpptrue.hpp".
3. Define an implementation class that inherits public virtual from foo::bar. e.g.

   ```
   class foo_bar_impl : public virtual foo(bar) { ... };
   ```
4. In the implementation class, provide a constructor that receives an instance handle and an `iluServer` as arguments, and calls the `iluObject` constructor appropriately, e.g.

```
foo_bar_impl::foo_bar_impl(char* pc_instance_handle, iluServer& r_an_ilu_server)
:
iluObject(foo(bar)::m_ILUClassRecord, pc_instance_handle, r_an_ilu_server)
{}
```

5. In the implementation class declaration, declare a virtual destructor. e.g. virtual foo_bar_impl::~foo_bar_impl();
6. Define the virtual destructor of the implementation class that (at the minimum) makes a call to iluDeactivate as the first thing it does, e.g.

```
foo_bar_impl::~foo_bar_impl() {
iluDeactivate();
// other app specific things that may need to be done
}
```

7. In the implementation class declaration, declare the virtual member functions that will implement the method(s), e.g.

```
virtual CORBA(Boolean) zap( CORBA(Long)  inarg, CORBA(Octet)&  inoutarg,
CORBA(Double)& outarg )
throw (CORBA(SystemException), foo(zapexception));
```

8. Define the virtual member functions that implement the method(s), e.g.

```
CORBA(Boolean) foo_bar_impl::zap( CORBA(Long)  inarg,
  CORBA(Octet)&  inoutarg,
  CORBA(Double)& outarg )
  throw (CORBA(SystemException), foo(zapexception)) {
  // do whatever must be done
  }
```

9. In for example 'main()', Call the runtime initialization function, passing an argument specifying whether or not to set up for threaded operation, e.g.

```
// Set up the runtime for threaded operation
iluCppRuntime::iluInitialize(ILUCPP_TRUE);
```

10. Create an `iluServer`. e.g.

```
iluServer server ("MyFooBarServerOnMyHost");
```

11. Create a true object in that server, e.g.

```
p_true_foo_bar = new foo_bar_impl("foo_bar_instance_0", server);
```

12. Publish the true foo::bar object, e.g.

```
p_true_foo_bar->iluPublish()
```

13. Run the server, e.g.

```
server.iluRun();
```

14. Compile your server application code, '`foo-cpp.cpp`' and '`foo-cpptrue.cpp`', and link the resulting object files with the C++ runtime library ('`libilu-cpp2.a`' on UNIX, and '`ilucpp2.lib`' on Win32) and the ILU kernel library ('`lib.a`' on UNIX, and '`ilu32.lib`' on Win32).

5.5.4 Client Basics

The basic steps in creating a simple client application are as follows (assuming we have a ISL file called 'foo.isl', describing an interface 'foo' with an object type 'bar'):

1. Run the C++ stubber on foo.isl, e.g. cpp2-stubber foo.isl
2. In your implementation file, e.g. 'clientmain.cpp', include the surrogate side header file, e.g. #include "foo-cppsurrogate.hpp".
3. In for example 'main()', call the runtime initialization function, passing an argument specifying whether or not to set up for threaded operation, e.g.

```
// Set up the runtime for threaded operation
iluCppRuntime::iluInitialize(ILUCPP_TRUE);
```

4. Lookup an object, e.g.

```
foo(bar_var) mybar_var = foo(bar)::iluLookup ("foo_bar_instance_0", "MyFooBarServe
```

5. Invoke a method, e.g.

```
try {
bool_return_value = mybar_var->zap(inarg, inoutarg, outarg);
}
catch (const foo(zapexception)& the_exception) {
/* do whatever */
}
catch (const CORBA(SystemException)& the_exception) {
/* do whatever */
}
catch (...) {
/* do whatever */
}
```

6. Compile your client application code, 'foo-cpp.cpp' and 'foo-cppsurrogate.cpp', and link the resulting object files with the C++ runtime library ('libilu-cpp2.a' on UNIX, and 'ilucpp2.lib' on Win32) and the ILU kernel library ('lib.a' on UNIX, and 'ilu32.lib' on Win32).

5.6 Relevant examples

The ILU examples directory contains two examples that use the CORBA 2.0 C++ mapping. See cpp2foo, and test1. The cpp2foo example illustrates a lot: object tables; collectible; custom surrogates; lookups; anys; return, in, inout, and out of most types; use of _vars; and more;

5.7 Runtime Classes

5.7.1 Overview

The classes of interest to the application programmer are listed below. Nearly all non-static member functions are virtual to allow creative overrides (at your own risk of course).

`iluCppRuntime` - Abstract class that provides various static member functions that the application can use to control the runtime's behavior.

`iluServer` - Provides the C++ view of a kernel server object.

`iluObject` - The most base class for all ILU C++ objects. All objects inherit either directly or indirectly from this class.

`iluObjectTable` - An abstract C++ class for developers to derive from to provide Object Tables.

`iluPassport` - encapsulates ilu_Passport functionality

`iluGSS` - encapsulates GSS functionality

`iluMainLoop` - An abstract base class for developers to derive from to create their own main loop.

`iluWString_var` Class - analog to CORBA(String_var) only for ILU Characters

(See '*ILUSRC*`/runtime/cpp2/ilu.hpp`' and '*ILUSRC*`/runtime/cpp2/corba.hpp`' for more complete descriptions.)

5.7.2 iluCppRuntime

An Abstract class that provides various static member functions that the application can use to control the runtime's behavior. `iluCppRuntime` is not meant to ever be subclassed.

5.7.2.1 iluCppRuntime - Initialization Related

`static void` **iluCppRuntime::iluInitialize** (`ILUCPP_BOOL` *b_use_native_threads* `= ILUCPP_FALSE`) C++

Initializes the C++ runtime for use. Also calls all the functions (typically interface initialization functions in generated stubs) that are on the C++ Runtime's initialization function list (see iluCppRuntime::iluAddInitializationFunction).

iluCppRuntime::iluInitialize's use depends on your use of threading:

1. No threading at all - just call iluCppRuntime::iluInitialize().
2. Using ILU's native operating system (OS) thread support - call iluCppRuntime::iluInitialize(ILUCPP_TRUE).
3. Your own thread package - call iluCppRuntime::iluSetForkProcedure, iluCppRuntime::iluSetNonNativeThreadIDFuncti then call the ILU kernel functions ilu_SetWaitTech, and ilu_SetLockTech appropriately, call iluMainLoop::iluSetMainLoop, then call iluCppRuntime::iluInitialize().

`static void` **iluCppRuntime::iluAddInitializationFunction** (`iluPFunctionInitializer` *pf_initialize*) C++

Adds an initialization function onto the runtime's list of (typically interface initialization) functions to call when iluCppRuntime::iluInitialize is called. iluPFunctionInitializer is typedeffed as

`void (*` **iluPFunctionInitializer**) ( ) C++

`static void` **iluCppRuntime::iluSetNonNativeThreadIDFunction** (`iluNonNativeThreadIDFunction` *p_thread_id_function*) C++

When running non-native threaded, this should be called (before initialization) set to the function that will return a thread unique `iluCardinal` id of the current thread. iluNonNativeThreadIDFunction is typedeffed as

`iluCardinal (*` **iluNonNativeThreadIDFunction**)( ) C++

`static iluPFunctionSurrogateCreator` **iluCppRuntime::iluSetSurrogateCreator** (`iluClass` *surrogate_class*, `iluPFunctionSurrogateCreator` *pfunction_surrogate_creator*) C++

Tells the C++ runtime what function to call when a surrogate for an object of the specified class is needed. This allows an implementation to subclass off a surrogate class, and write a new surrogate creation function that creates an instance of this new subclass. This more specialized surrogate might do message filtering, caching, monitoring, etc. Call this function after you've performed initialization, but before you do any operations which might create a surrogate of the specified class. It basically overwrites the default surrogate creation function set up by the surrogate stubs. It returns the old surrogate creator function, or `NULL` if was previously no surrogate creator for that class (note: `NULL` return should not really happen unless a mistake or something clever is being done - this means you've added a new node to the surrogate creator function list). iluPFunctionSurrogateCreator is typedeffed as

`iluObject* (*` **iluPFunctionSurrogateCreator**) (`iluKernelObject`). C++

A surrogate creator function should at the minimum create an instance of a surrogate, call the instances member function iluAssociateKernelObject passing the `iluKernelObject,` and then return a pointer to the new instance.

`static void` **iluCppRuntime::iluSetForkProcedure** (`iluForkProc` *pfunction_fork_procedure*) C++

If your using your own threads package call this before calling the ILU kernel functions ilu_SetWaitTech, etc. and pass a pointer to your function that forks a thread. iluForkProc is typedeffed as

`iluBoolean (*` **iluForkProc**) (`void (*pfunction_procedure) (void*` *pv_argument*`),`
void* *pv_argument*, ILU_ERRS((no_memory, no_resources, internal)) * *p_error*) C++

5.7.2.2 iluCppRuntime - Character Utilities

`static iluCardinal` **iluCppRuntime::iluCharacterStringLength** (`const iluCharacter*` *p_chars*) C++

Returns the length of the `iluCharacter` string

`static iluCharacter*` **iluCppRuntime::iluCharacterStringCopy** (`iluCharacter*` *p_chars_destination*, `const iluCharacter*` *p_chars_source*) C++

Copies the source `iluCharacter` string to the destination, returns the destination.

`static iluCharacter*` **iluCppRuntime::iluCharacterStringDuplicate** (`const iluCharacter*` *p_chars_source*) C++

Returns a duplicate of the source `iluCharacter` string

`static ILUCPP_BOOL` **iluCppRuntime::iluCharacterStringEqual** (`const iluCharacter*` *p_chars_one*, `const iluCharacter*` *p_chars_two*) C++

Returns true if strings are the same, else false.

`static iluCharacter*` **iluCppRuntime::iluCharStringFromShortCharString** (`const iluShortCharacter*` *pc_shortchars*) C++

Returns a new `iluCharacter` string filled in from the `iluShortCharacter` string.

`static ILUCPP_BOOL` **iluCppRuntime::iluCharStringShortCharStringEqual** (`const iluCharacter*` *pc_chars*, `const iluShortCharacter*` *pc_shortchars*) C++

Returns true if the `iluCharacter` string matches the `iluShortCharacter` string.

5.7.2.3 iluCppRuntime - String Binding Handle Utilities

`static char*` **iluCppRuntime::iluFormSBH** (`const char*` *pc_serverid*, `const char*` *pc_instance_handle*, `iluClass` *the_ilu_class*, `iluProtocolInfo` *pc_protocol_type* = ((`iluProtocolInfo`) `NULL`), `iluTransportInfo` *transport_info* = ((`iluTransportInfo`) `NULL`)) C++

`static char*` **iluCppRuntime::iluFormSBHUsingContactInfo** (`const char*` *pc_serverid*, `const char*` *pc_instance_handle*, `iluClass` *the_ilu_class*, `const char*` *p_str_encodedContactInfo* = `NULL`) C++

Use these to form a string binding handle from relevant parts, if protocol and/or transport info are NULL, current defaults are used. For iluFormSBHUsingContactInfo, p_str_encodedContactInfo is as would be obtained from iluParseSBH.

static ILUCPP_BOOL **iluCppRuntime::iluParseSBH** (iluCString *str_encodedSBH*, iluCString* *p_str_plainInstanceHandle* = NULL, iluCString* *p_str_plainServerID* = NULL, iluCString* *p_str_plainMstid* = NULL, iluCString* *p_str_encodedContactInfo* = NULL, iluCardinal* *p_card_encodedContactInfoLen* = NULL, ILUCPP_BOOL* *p_b_malloced_contact_info* = NULL)) C++

Parse a string binding handle, returning whichever elements are specified by passing in non-NIL pointers. Caller retains ownership of URL argument. If p_str_plainInstanceHandle != NIL, ownership of *p_str_plainInstanceHandle is passed to caller iff successful. Similarly for p_str_plainServerID and p_str_plainMstid. *p_str_encodedContactInfo is set to point into the given URL (The whole sequence of contact info is returned in *p_str_encodedContactInfo) , and *p_card_encodedContactInfoLen is set to the length of the contact info substring; the next character is left unmolested. If the p_b_malloced_contact_info out parameter is true, then caller must arrange to free it.

5.7.2.4 iluCppRuntime - File Descriptor Budget

static iluCardinal **iluCppRuntime::iluGetFDBudget**( ) C++

static iluCardinal **iluCppRuntime::iluSetFDBudget** (iluCardinal *card_size*) C++

Get and set ILU file descriptor budget. iluSetFDBudget returns the new budget. Because ILU may open multiple connections to a server, we need some policy for when to close them. That policy is this: the application gives the ILU kernel a "File Descriptor Budget" (initally 16). The ILU kernel promises to use no more than this many File Descriptors at once.Off the top of this budget we take FDs needed for serving (one per listening socket and one per accept). The remainder is allocated to outgoing connections (over transports that use FDs --- ie, not inmemory). When we want to consume a new FD, and there's no room left in the budget, we go looking for an idle outgoing connection (one with no outstanding calls) to close. All idle outgoing connections are kept in a doubly-linked list, ordered by when the connection went idle (most recently at the front).

5.7.2.5 iluCppRuntime - Memory Management

static void **iluCppRuntime::iluFree** (void* *pv* /* ILUowned */) C++

Use this to free things returned by ILU

static void* **iluCppRuntime::iluMalloc** (iluCardinal *card_size*) C++

You can use this to malloc things from ILU.

5.7.3 iluServer

`iluServer` provides a the C++ view of a kernel server object. `iluServers` cannot be copied or assigned.

5.7.3.1 iluServer - Setup and Destruction

`iluServer::iluServer` (`char*` *pc_server_id* = `NULL`, `iluObjectTable*` *p_object_table* = `NULL` `/* ILUowned */` , `char *` *pc_protocol_type* = `NULL`, `iluTransportInfo` *transport_info* = `NULL`, `iluPassport*` *p_passport* = `NULL`) C++

Constructor - If no pc_server_id is specified, one is automatically created based on based on time, hostname, and process id. If p_object_table is `NULL`, a default object table implementation is used. A port is created and added to the server using the specified protocol and transport, and becomes the default port of the server. pc_protocol_type and transport_info default to whatever the default protocol and transport are currently set to. Caller owns pc_server_id p_object_table, pc_protocol_type, transport_info, and p_passport. p_passport points to an `iluPassport,` defaulted to `NULL` - this passport containing an ILU GSS identity, which is used as the identity of the principal offering the service, and put into the connection information in the string binding handle of objects on that server.

`virtual` **iluServer::~iluServer** ( ) C++

Destructor - basically destroys the kernel server and breaks all associations between kernel objects in this server and their language specific objects. Indirectly also deletes any `iluObjectTable` used with this iluServer.

`virtual void` **iluServer::iluAddPort** (`char*` *pc_protocol_type*, iluTransportInfo *transport_info*, ILUCPP_BOOL *b_become_default_port* = `ILUCPP_FALSE`, iluPassport* *p_passport* = `NULL`) C++

Adds another port to an existing server If b_become_default_port is `ILUCPP_TRUE` the new port will become the default port for this server. p_passport points to an `iluPassport,` defaulted to `NULL`. this passport containing an ILU GSS identity, which is used as the identity of the principal offering the service, and put into the connection information in the string binding handle of objects on that server. Caller owns the arguments.

`virtual void` **iluServer::iluRun** (`int*` *p_i_stop_on_non_zero* = `NULL`) C++

This runs the main, outer loop of an iluServer. It never returns if p_i_stop_on_non_zero isn't supplied, else it returns when *p_i_stop_on_non_zero is non zero. If you're running threaded this routine simply goes into a sleep loop,

5.7.3.2 iluServer - Default Accessors

`static char*` **iluServer::iluGetDefaultProtocol** ( ) C++

`static void` **iluServer::iluSetDefaultProtocol**(`char*` *pc_new_default_protocol*) C++

Get and set the default protocol used when adding a port on a `iluServer` - initialized to whatever is set to be the default in the kernel (found in '*ILUSRC*`/runtime/kernel/iluconf.h`' or '*ILUSRC*`/runtime/kernel/iluwin.h`') `static const iluTransportInfo` **iluServer::iluGetDefaultTransport** ( ) C++

`static void` **iluServer::iluSetDefaultTransport(** `iluTransportInfo` *ppc_new_default_transport_info*) C++

Get and set the default transports used when adding a port on a `iluServer` - initialized to whatever is set to be the default in the kernel (found in '*ILUSRC*`/runtime/kernel/iluconf.h`' or '*ILUSRC*`/runtime/kernel/iluwin.h`') Callee owns pc_new_default_transport_info.

`static iluServer&` **iluServer::iluGetDefaultServer** ( ) C++

Returns the default `iluServer`, creating one if need be.

`static iluServer*` **iluServer::iluSetDefaultServer** (`iluServer&` *new_default_server*) C++

Sets the default `iluServer`, returns old default, which is `NULL` if no default currently is set.

5.7.4 iluObject

The most base class for all ILU C++ objects. All objects inherit either directly or indirectly from `iluObject`. All non-static member functions are virtual to allow creative overrides (at your own risk of course). `iluObjects` cannot be copied or assigned.

5.7.4.1 iluObject - Creation and Destruction

`iluObject::iluObject` (`iluClass` *the_Class*, `char*` *pc_instance_handle* = `NULL`, `iluServer&` *the_server* = `iluServer::iluGetDefaultServer()`, **ILUCPP_BOOL** *b_within_object_table* **= ILUCPP_FALSE)** C++

Constructor - This constructor must be called (only) from the constructors for true objects.

For example, in an implementation of a foo::bar :

```
  foo_bar_impl(char* pc_instance_handle, iluServer& r_an_ilu_server, CORBA(Boolean)
b_within_object_table = ILUCPP_FALSE :
          iluObject(foo(bar)::m_ILUClassRecord, pc_instance_handle, r_an_ilu_server,
b_within_object_table) {}
```

If no instance handle is specified, then the value of a monotonicaly increasing, `iluServer` specific counter will be used to generate one. If no server is specified, then the default server will be used. (The default server is generated automatically if needed, and has the an id based on time, hostname, and process id.) Caller owns pc_instance_handle. The new object has a reference count of 1. If b_within_object_table is true, then it is assumed the object is being created inside an iluObjectTable's iluObjectOfInstanceHandle function, meaning that the locks on the server should not be modified.

`static iluObject*` **iluObject::iluStringToObject** (`char*` *pc_string_binding_handle*) C++

Given a string binding handle (e.g. as obtained from iluObjectToString) returns an iluObject* for that object, with the reference count incremented.

`virtual` **iluObject::~iluObject** ( ) C++

Destructor ensures that this object is completely disassociated from the ILU kernel The most specific destructor of an object should call iluDeactivate on the object to block any further incoming calls, and wait for any ongoing calls to complete. Next it should perform any object specific cleanup. Finally, the destructor in `iluObject` will break the association between the kernel object and this object, allowing the kernel object to be potentially freed.

`virtual void` **iluObject::iluDeactivate** ( ) C++

Ensures this object is not available from the outside. This must be the first thing called by the most specific destructor of an object. If if isn't, then the potential exists (in multithread case) for a call to come in for an object that's in the middle of destruction - a bad thing! This function blocks until there are zero ongoing calls.

`virtual void` **iluObject::iluKernelObjectUnlinked** ( ) C++

Called by iluUnlinkKernelObject - you can override this virtual function in your objects to do whatever you like when the association between your object and the kernel object is broken - e.g. delete yourself The implementation in `iluObject` deletes this.

5.7.4.2 iluObject - Object Publication

`virtual ILUCPP_BOOL` **iluObject::iluPublish** ( ) C++

Publishes binding information for this object in the binding service Has no effect on object reference count.

`virtual ILUCPP_BOOL` **iluObject::iluWithdraw** ( ) C++

Removes binding information for this object from the binding service Has no effect on object reference count.

`static void*` **iluObject::iluLookup** (`char*` *pc_instance_handle*, `char*` *pc_server_id*, `iluClass` *the_class*) C++

Used by stubber generated iluLookup functions in derived classes to lookup an object in the binding service based on its instance and server id and class. Increments reference count of object. To Lookup objects of type T, use the T::iluLookup(char* *pc_instance_handle*, char* pc_server_id; function produced by the stubber. For example:

```
mybar_var =  foo(bar)::iluLookup("foo_instance_0", pc_serverid )
```

5.7.4.3 iluObject - Accessors

`virtual iluServer*` **iluObject::iluGetServer** `()` C++

Returns pointer to the iluServer that this object resides in.

`virtual const char* /* ILUowned */` **iluObject::iluId** `()` C++

Returns the objects instance id.

`virtual const char* /* ILUowned */` **iluObject::iluServerId** `()` C++

Returns the id of the objects ILU Server.

5.7.4.4 iluObject - Informational

`virtual iluCString` **iluObject::iluObjectToString** `()` C++

Returns the ILU string binding handle for the object. Caller get ownership of the string

`ILUCPP_BOOL` **iluObject::iluIsCollectibleObject** `()` C++

Returns true if the object is of a collectible class.

`virtual iluCString` **iluObject::iluObjectToIORString** `()` C++

Returns a string which is the object's name and contact information as specified by the CORBA IIOP spec - caller gets ownership of the string. May return NULL if the object is not exported through an `IIOP` ilu_Port. (Available only when `IIOP` support is configured into ILU.)

`virtual iluCString` **iluObject::iluObjectToURLString** `()` C++

Returns a string which is the object's name and contact information as specified by an `HTTP` URL - caller gets ownership of the string. May return NULL if the object is not exported through an `HTTP` ilu_Port (Available only when `HTTP` support is configured into ILU.)

`ILUCPP_BOOL` **iluObject::iluPing()** C++

Returns `ILUCPP_TRUE` if the true object exists, and the process serving it can be contacted, otherwise `ILUCPP_FALSE`.

`ILUCPP_BOOL` **iluObject::_is_equivalent(iluObject*** `p_obj`) C++

Returns `ILUCPP_TRUE` if the two objects denote the same thing.

`ILUCPP_BOOL` **iluObject::iluInSameServer(iluObject*** `p_obj`) C++

Returns `ILUCPP_TRUE` if the two objects are in the same ILU server. Used, for example, to determine if objects are SIBLINGS.

`virtual const char* /* ILUowned */` **iluObject::iluClassName** ( ) C++

`virtual const char* /* ILUowned */` **iluObject::iluClassId** ( ) C++

Return the ILU class name and type id - primarily informational use.

5.7.4.5 iluObject - Reference Counting

`virtual void` **iluObject::iluIncrementReferenceCount**( ) C++

`virtual void` **iluObject::iluDecrementReferenceCount**( ) C++

Reference count operations - when an object is first created, it has a reference count of one. If the reference count ever goes to zero, delete is called on this. CORBA compliant apps (where objects derive from CORBA::Object) should use the duplicate and release functionality defined in the CORBA specification.

`virtual iluCardinal` **iluObject::iluGetReferenceCount**( ) C++

Returns what the current reference count is.

`static iluObject*` **_duplicate(iluObject*** *p_obj*) C++

Increments the reference count on the object and returns it. Returns NULL if passed NULL.

`static iluObject*` **_narrow(iluObject*** *p_obj*) C++

Effectively casts the object pointer to an iluObject*.

5.7.5 iluObjectTable

An abstract C++ class for developers to derive from to provide Object Tables. Object tables cannot be copied or assigned. Besides doing whatever application specific things might need to be done in the constructor and destructor, a class derived from `iluObjectTable` must provide the iluObjectOfInstanceHandle virtual member function.

`virtual iluObject*` **iluObjectTable::iluObjectOfInstanceHandle** C++
(`iluCString` *pc_instance_handle* `/* ILUowned */`) = `0;`

Called by ILU to create and return a new `iluObject*` with the specified instance handle. ILU retains ownership of pc_instance_handle - i.e. copy it if you need want to hang on to it. Note that when in this function, you are 'inside' the object's server - i.e. you hold the locks on the server - this means that when you create the object, you must specify the 3rd argument to the object's constructor (b_within_object_table) as true.

`virtual iluServer*` **iluObjectTable::iluGetServer** `();` C++

Returns the pointer to the iluServer this object table is associated with.

`virtual` **iluObjectTable::~iluObjectTable** ( ) C++

Do whatever destroying the Object Table needs to do to free up resources, etc. It gets called when the `iluServer` it's associated with it is shut down.

5.7.6 iluPassport

Encapsulates `ilu_Passport` functionality

`iluPassport::iluPassport` (`iluIdentityInfo` *p_identity_info* = `NULL`) C++

Constructor - creates and returns a passport, optionally containing the specified identity.

`virtual` **iluPassport::~iluPassport** ( ) C++

Destructor - frees any associated identities in addition to freeing the passport

`static iluPassport*` **iluPassport::iluGetPassport** ( ) C++

`static iluPassport*` **iluPassport::iluSetPassport** (`iluPassport*` *p_passport*) C++

Get and set the passport being used for outgoing calls - in the multi-threaded case, this is per-thread Set returns the old `iluPassport`. Note that before your thread exits, you should either call iluSetPassport(`NULL`), or delete the `iluPassport` in use (assuming it's only in use for a single thread).

`static iluPassport* /* ILUowned */` **iluPassport::iluGetCallerPassport**( ) C++

Get the passport of the caller.

`virtual void` **iluPassport::iluAddIdentity** (`iluIdentityInfo` *p_identity_info* `/* ILUowned */`) C++

Adds identity to Passport. Only one identity of each type is allowed.

`virtual iluIdentityInfo /* ILUowned */` **iluPassport::iluFindIdentity** (`iluIdentityType` *p_identity_type*) C++

Returns identity of specified type, if present else `NULL`

`static iluIdentityInfo` **iluPassport::iluCopyIdentity** (`iluIdentityInfo` *p_identity_info*) C++

Returns a copy of the passed identity

`ilu_Passport /* ILUowned */` **iluPassport::iluGetIluPassport** () C++

Returns the (kernel) ilu_Passport

5.7.7 iluGSS

Encapsulates GSS functionality - only defined when ILU is configured with secure transport.

`static iluIdentityInfo` **iluGSS::iluAcquireGSSIdentity** (`gss_cred_id_t` *gss_credential*) C++

`static ILUCPP_BOOL` **iluGSS::iluDecodeGSSIdentity** (`iluIdentityInfo` *p_identity_info*, `gss_name_t*` *p_name*, `iluFineTime*` *p_good_till_time*, `gss_OID` *mechanism*, `ILUCPP_BOOL*` *p_b_local*, `iluCardinal*` *p_card_flags*) C++

p_identity_info - input; retain; info to decode *p_name* - output; name in identity *p_good_till_time* - output; good-till *mechanism* - input; actual mechanism desired; optional *p_b_local* - if TRUE, local; otherwise remote *p_card_flags* - connection flags, as in gss_inquire_context

`static gss_cred_id_t` **iluGSS::iluAcquireGSSCredForName** (`char*` *pc_name*, `iluCardinal` *card_lifetime*, `gss_OID` *mechanism*, `ILUCPP_BOOL` *b_accept_only*) C++

`static iluCString` **iluGSS::iluGSSNameToString** (`gss_name_t` *name*) C++

5.7.8 iluMainLoop

Subclass from the `iluMainLoop` class if you want to have your own version of the main loop. `iluMainLoops` cannot be copied or assigned. A single threaded application should supply all functions. An application making use of ILU's OS multi-threaded operation should not use a different mainloop. If you're using your own thread package, you must supply all functions, and see the comment for iluCppRuntime::iluInitialize

`virtual void` **iluMainLoop::iluRun** (`int*` *p_i_stop_on_non_zero*) `= 0;` C++

Runs the main loop until *p_i_stop_on_non_zero is non-zero.

`virtual void` **iluMainLoop::iluExit** (`int*` *p_i_stop_on_non_zero*) `= 0;` C++

Causes the main loop to exit

`virtual ILUCPP_BOOL` **iluMainLoop::iluRegisterInputHandler** (`int` *i_fd*, void (* *pfunction_input_handler*)(int *i_fd*, void* *pv_input_handler_arg*), void* *pv_input_handler_arg*) = 0; C++

Input Handlers - When there is input activity on the file descriptor i_fd, the mainloop will call the registered handler procedure pfunction_input_handler, passing it i_fd and pv_input_handler_arg as arguments. Returns `ILUCPP_FALSE` if it can't do it's job due to some resource limitation.

`virtual ILUCPP_BOOL` **iluMainLoop::iluUnregisterInputHandler** (`int` *i_fd*, void (** *ppfunction_input_handler*)(int *i_fd*, void* *pv_input_handler_arg*), void** *ppv_input_handler_arg*) = 0; C++

Returns `ILUCPP_FALSE` if input on i_fd was being handled, else `ILUCPP_TRUE`. Sets function and arg ptrs to what they were if anything.

`virtual ILUCPP_BOOL` **iluMainLoop::iluRegisterOutputHandler** `(int` *i_fd*`,` `void` (* *pfunction_output_handler*)(`int` *i_fd*, `void*` *pv_output_handler_arg*), `void*` *pv_output_handler_arg*) = `0;` C++

Output Handlers - When it is possible to perform output on the file descriptor i_fd, the mainloop will call the registered handler procedure pfunction_output_handler, passing it i_fd and pv_output_handler_arg as arguments.

`virtual ILUCPP_BOOL` **iluMainLoop::iluUnregisterOutputHandler** (`int` *i_fd*, void (** *ppfunction_output_handler*)(int *i_fd*, void* *pv_output_handler_arg*), void** *ppv_output_handler_arg*) = 0; C++

Returns `ILUCPP_FALSE` if output on i_fd had a handler, else `ILUCPP_TRUE` Sets function and arg ptrs to what they were if anything.

`virtual iluAlarm` **iluMainLoop::iluCreateAlarm** ( ) = `0;` C++

Creates an alarm. An alarm is an active object which can be set to asynchronously invoke a procedure with an argument at a specified time. An alarm may be something like a pointer to a structure that has some internal structure, but from the point of view of an alarm user, it's just a handle that is used to specify a particular alarm to be set or cleared.

`virtual void` **iluMainLoop::iluSetAlarm(iluAlarm** *the_alarm*, `iluFineTime` *alarm_time*, `void` (**pfunction_alarm_handler*)(`void*` *pv_alarm_handler_arg*), `void*` *pv_alarm_handler_arg*) = `0;` C++

Sets up an alarm to call the handler procedure pfunction_alarm_handler, passing it pv_alarm_handler_arg as an argument, when the alarm_time occurs.

`virtual void` **iluMainLoop::iluClearAlarm** (`iluAlarm` *the_alarm*) = `0;` C++

Cancels the alarm (effectively sets the alarm time to infinity).

`virtual void` **iluMainLoop::iluDestroyAlarm** (`iluAlarm` *the_alarm*) = `0;` C++

Destroys the alarm (if alarm is set, does not invoke).

`static void` **iluMainLoop::iluSetFineTimeFromNow(** `ilu_FineTime*` *p_finetime*, `ilu_integer` *i_secs*, `ilu_cardinal` *i_msecs*) C++

Utility function to set the pointed to ilu_FineTime to a time i_secs + i_msecs in the future

`static void` **iluMainLoop::iluSetMainLoop** (`iluMainLoop*` *p_mainloop_instance*) C++

`static iluMainLoop*` **iluMainLoop::iluGetMainLoop** ( ) C++

Setting the Main Loop to be used - Call iluSetMainLoop set your mainloop as the one for ILU to use. It should called before any ILU initialization.

`static iluAlarm` **iluMainLoop::iluDefaultLoopCreateAlarm** ( ) C++

`static void` **iluMainLoop::iluDefaultLoopSetAlarm** (`iluAlarm` *the_alarm*, `iluFineTime` *alarm_time*, `void` (* *pfunction_alarm_handler*)(`void*` *pv_alarm_handler_arg*), `void*` *pv_alarm_handler_arg*) C++

`static void` **iluMainLoop::iluDefaultLoopClearAlarm** (`iluAlarm` *the_alarm*) C++

`static void` **iluMainLoop::iluDefaultLoopDestroyAlarm** (`iluAlarm` *the_alarm*) C++

When you haven't set the main loop (i.e. you're using ILU's default loop), you can call these functions to create, set and unset alarms. (If you set your own main loop, just call its alarm functions.)

5.8 CORBA 2.0 C++ Considerations

5.8.1 ORB_init

The CORBA `ORB_init` function may be called instead of `iluCppRuntime::iluInitialize` (which `ORB_init` calls internally). The orb identifier passed to `ORB_init` should be `"ilu"`. If the command line argument `-iluthreaded` is present in the command line arguments passed to `ORB_init`, then ILU will be run in threaded mode, otherwise ILU will run single-threaded.

6 Using ILU with C++

6.1 Introduction

This document is for the C++ programmer who wishes to use ILU. The following sections will show how ILU is mapped into C++ constructs and how both C++ clients and servers are generated and built.

When functions are described in this section, they are sometimes accompanied by *locking comments*, which describe the locking invariants maintained by ILU on a threaded system. See the file '*ILUSRC*`/runtime/kernel/iluxport.h`' for more information on this locking scheme, and the types of locking comments used.

A number of macros are used in function descriptions, to indicated optional arguments, and ownership of potentially malloc'ed objects. The owner is responsible for freeing the object's storage at an appropriate time, and generally makes no interesting guarantees of when that will be. Some types of objects (generally (fixed- and variable-length) arrays, including strings) are presumed to be mutable by only their owners. The macro `OPTIONAL(`*type-name*`)` means that the value is either of the type indicated by *type-name*, or the value `NULL`. This macro may only be used with pointer values. The macro `RETAIN(`*type-name*`)` indicates, when used on a parameter type, that the caller retains ownership of the value, and when used on a return type, that the called function retains ownership of the value. The macro `PASS(`*type-name*`)` indicates, when used on a parameter type, that the caller is passing ownership of the storage to the called function, and when used on a return type, that the called function is passing ownership of the called value to the caller. The macro `GLOBAL(`*type-name*`)` means that neither the caller nor the calling function owns the storage.

6.2 Mapping ILU ISL to C++

Using ILU with C++ is intended to eventually be compatible with the OMG CORBA specification. That is, all of the naming and stub generation comply with the Common Object Request Broker Architecture specified mapping for C++, when that specification is available. The current mapping was designed to be usable with a large number of C++ compilers, by avoiding problematic constructs such as templates, exceptions, namespaces, and nested class definitions.

Note that ILU support for C++ does rely on having argument prototypes, all C++ library functions, and the capabilities of the C++ pre-processor.

6.2.1 Names

In general, ILU constructs C++ names from ISL names by replacing hyphens with underscores. Type names are prepended with their interface name and the string ``_T_''. Enumeration value namess are formed by prepending the enumeration type name and ``_'' to the ISL enumeration value name. Exception names are prepended with their interface name and ``_E_''. Constant names are prepended with their interface name and ``_C_''.

Other naming conventions may be specified explicitly; see the following section on tailoring names for more information.

6.2.2 Types

Records turn directly into C++ structs. Unions consist of a struct with two fields: the type discriminator, a field called ‘‘discriminator’’, and a union of the possible values, called ‘‘value’’. Arrays map directly into C++ arrays. Sequences become a C++ class with methods and representation analogous to the procedures and representation that appear in the C mapping. Objects become normal C++ classes that are subclasses of the pre-defined class `iluObject`.

6.2.2.1 Sequence types

For most sequences types, the generated C++ code follows a pattern analogous to that set in the C mapping and illustrated in the upcoming example.

6.2.3 Object types

6.2.4 Exceptions

Because of the scarcity of implementation of the C++ exception mechanism, exceptions are passed by adding an additional argument to the beginning of each method, which is a pointer to a status struct, which contains an exception code, and a union of all the possible exception value types defined in the interface. Method implementations set the exception code, and fill in the appropriate value of the union, to signal an exception. Exception codes are represented in C++ with values of the type `ilu_Exception`.

In a true module, exceptions may be raised by using the function <*interface*>`_G::RaiseException`.

`void` <*interface*>**_G::RaiseException** (`RETAIN(`<*interface*>`Status *)` *status*, `GLOBAL(ilu_Exception)` *code*, `...`) C++

Causes an exception code and value for the exception specified by *code* to be bound in *status*. Besides the two required arguments, the function may take another argument, which should be a value of the type implied by the value of *code*; that is, of the appropriate type to be a value of the exception being signalled. Note that `RaiseException` does not actually cause a transfer of control, so that an explicit return statement must follow a call to `RaiseException`.

6.2.5 Constants

Constants are implemented with C++ `#define` statements.

6.2.6 Examples

Here’s a sample ISL spec, and the resulting C++ mappings:

```
INTERFACE Foo;

TYPE String = ilu.CString;
TYPE UnsignedInt = CARDINAL;

TYPE E1 = ENUMERATION val1, val2, val3=40 END;
TYPE R1 = RECORD field1 : CARDINAL, field2 : E1 END;
TYPE A1 = ARRAY OF 200 BYTE;
TYPE A2 = ARRAY OF 41, 3 R1;
TYPE S1 = SEQUENCE OF E1;
TYPE U1 = UNION R1, A2 END;

EXCEPTION Except1 : String;

CONSTANT Zero : CARDINAL = 0;

TYPE O1 = OBJECT
  METHODS
    M1 (arg1 : R1) : UnsignedInt RAISES Except1 END
  END;
```

The C++ mapping:

```
typedef ilu_CString Foo_T_String;
typedef ilu_Cardinal Foo_T_UnsignedInt;

typedef enum _Foo_T_E1_enum {
  Foo_T_E1_val1 = 1,
  Foo_T_E1_val2 = 2,
  Foo_T_E1_val3 = 40
} Foo_T_E1;
typedef struct _Foo_T_R1_struct {
  ilu_Cardinal field1;
  Foo_T_E1 field2;
} Foo_T_R1;
typedef ilu_Byte Foo_T_A1[200];
typedef Foo_T_R1 Foo_T_A2[41][3];
class _Foo_T_S1_sequence {
 private:
  ilu_Cardinal _maximum;
  ilu_Cardinal _length;
  Foo_T_E1 *_buffer;
 public:
  _Foo_T_S1_sequence ();
  virtual ~_Foo_T_S1_sequence ();
  static class _Foo_T_S1_sequence *Create (ilu_Cardinal initial_size, Foo_T_E1
*initial_data);
  virtual ilu_Cardinal Length();
  virtual void Append(Foo_T_E1);
  virtual Foo_T_E1 RemoveHead();
```

```
    virtual Foo_T_E1 RemoveTail();
    virtual ilu_Cardinal RemoveAll(ilu_Boolean (*matchproc)(Foo_T_E1));
    virtual Foo_T_E1 * Array();
    virtual Foo_T_E1 Nth(ilu_Cardinal index);
  };
  typedef class _Foo_T_S1_sequence * Foo_T_S1;
  enum Foo_T_U1_allowableTypes { Foo_T_U1_R1, Foo_T_U1_A2 };
  typedef struct _Foo_T_U1_union {
    enum Foo_T_U1_allowableTypes discriminator;
    union {
      Foo_T_R1 R1;
      Foo_T_A2 A2;
    } value;
  } Foo_T_U1;

  extern ilu_Exception Foo_E_Except1;     /* exception code Except1 */

  typedef struct _Foo_Status_struct {
    ilu_Exception returnCode;
    union {
      ilu_Cardinal anyvalue;
      Foo_T_String Except1;
    } values;
  } FooStatus;

  class Foo_T_O1 : public iluObject {
   public:
    Foo_T_O1(); // constructor
    virtual ~Foo_T_O1(); // destructor
    static class Foo_T_O1 * ILUCreateFromSBH(ilu_CString sbh);
    Foo_T_UnsignedInt M1 (FooStatus *_status, Foo_T_R1 *arg1);
  };

  #define Foo_C_Zero ((ilu_Cardinal) 0)
```

6.3 Using an ILU module from C++

A client module may obtain an instance of an ILU object in three basic ways: 1) instantiating it directly from a string binding handle, 2) using the function `iluObject::Lookup` to locate it via the simple binding interface, and 3) receiving the instance directly as a return value or out parameter from a method on a different object.

To instantiate from a string binding handle, a static member function is generated for each subclass of `class iluObject` declared in the C++ stubs:

`OPTIONAL(class` *T* `*)` ***T*::ILUCreateFromSBH** (`ilu_CString` *sbh*) C++

To use the simple binding service to locate an object:

static OPTIONAL(GLOBAL(void *)) **iluObject::Lookup** (RETAIN(char *) *sid*, RETAIN(char *) *ih*, ilu_Class *putative-class*) C++

Locking: Main invariant holds.

Finds and returns the object specified by the given Server ID (*sid*) and server-relative Instance Handle (*ih*) by consulting the local domain registry of objects. *putative-class* is the type that the object is expected to be of, though the type of the actual object returned may be a subtype of *putative-class*, cast to the *putative-class*. The return value should be immediately cast to a value of the C++ mapping of *putative-class*.

6.4 Implementing an ILU Module in C++

For each ILU class *interface*.*otype*, ILU will define, in the file '*interface*.cc', a C++ class called *interface*_T_*otype*. To implement a true object for *interface*.*otype*, one should further subclass this C++ class, and override all of its methods. In particular, do not let any of the default methods for the class be called from your methods for it.

6.4.1 Servers

ILU supports, in each address space, multiple instances of something called a *kernel server*, each of which in turn supports some set of object instances. A kernel server *export*s its objects by making them available to other modules. It may do so via one or more *port*s, which are abstractly a tuple of (*rpc protocol*, *transport type*, *transport address*). For example, a typical port might provide access to a kernel server's objects via (Sun RPC, TCP/IP, (host 13.24.52.9, UNIX port 2076)). Another port on the same kernel server might provide access to the objects via (Xerox Courier, XNS SPP, XNS port 1394).

When creating an instance of a true object, a kernel server for it, and an instance handle (the name by which the kernel server knows it) for it must be determined. These may be specified explicitly by overriding the default iluObject::ILUGetServer and iluObject::ILUGetInstanceHandle methods, respectively. The iluObject implementation of ILUGetServer defers to ilu::GetDefaultServer. The iluObject implementation of ILUGetInstanceHandle generates a handle that's unique relative to the kernel server.

The kernel server is represented in C++ with the class iluServer, which has the following constructor:

?? **iluServer::iluServer** (OPTIONAL(const char *) *server-id*, OPTIONAL(iluObjectTable *) *object-table*) C++

Constructs an instance of class iluObject with the given *server-id* and *object-table*.

Note that ILU object IDs, which consist of the kernel server ID, plus the instance handle of the object on that server, must be unique ''across space and time'', as the saying goes. If no kernel server id is specified, ILU will generate one automatically, using an algorithm that provides a high probability of uniqueness. If you explicitly specify a kernel server ID, a good technique is to use a prefix or suffix which uniquely identifies some domain in which you can assure the uniqueness of the remaining part of the ID. For example, when using ILU at some project called NIFTY at some internet site in the IP domain `department.company.com`, one might use kernel server IDs with names like `something.NIFTY.department.company.com`.

Once the server is constructed, a port must be added:

`ilu_Boolean` **iluServer::AddPort** (`OPTIONAL`(`RETAIN`(`char *`)) *protocol-info*, `OPTIONAL`(`RETAIN`(`ilu_TransportInfo`)) *transport-info*, `ilu_Boolean` *be-default*) C++

Adds a port through which the server can be contacted. The *protocol-info* and *transport-info* specify the RPC and transport protocols and their parameters. The *transport-info* has a layered structure, represented by the C type `ilu_TransportInfo`, described in '*ILUSRC*`/runtime/kernel/iluxport.h`'. See chapter 1 for a catalogue of available RPC and transport layer specifications.

To export a module for use by other modules, simply instantiate one or more instances of your subtype of *interface*`:`*otype* and (if single-threaded) call the ILU C++ event dispatching loop, `iluServer::Run`.

6.4.2 Event dispatching

Most non-threaded long-lived C and C++ programs simulate threads with *event dispatching*, in which the program waits in some piece of code called the *main loop* until an *event* such as input arriving on a file descriptor or the expiration of an alarm signal causes a *callback routine* to be invoked. The ILU C++ runtime, in single-threaded mode, supports this style of operation with various static member functions of the class `iluServer`.

`static ilu_Boolean` **iluServer::RegisterInputHandler** (`int` *fd*, `void` (`*`*callbackRoutine*)(`int, void *`), `void *` *callbackArg*) C++

Register the file descriptor *fd* with the ILU kernel so that when ILU kernel event dispatching is active (that is, during the `iluServer::Run` call), the function *callbackRoutine* will be invoked with the arguments `(`*fd*`,` *callbackArg*`)` whenever input is available on the file descriptor *fd*.

`static ilu_Boolean` **iluServer::UnregisterInputHandler** (`int` *fd*) C++

Removes any callback routine registered on file descriptor *fd*.

`static ilu_Boolean` **iluServer::Run** (`void`) C++

Invokes the ILU main loop and causes ILU kernel event dispatching to be active. This routine never returns.

Occasionally it is necessary to use a different event dispatching mechanism, typically because some other work is done inside the main loop of the mechanism. An alternate main loop can be registered for use with ILU by creating a subtype of the class `iluMainLoop` and registering it with the kernel by calling the function `iluServer::iluSetMainLoop`:

static void **iluServer::iluSetMainLoop** (RETAIN(iluMainLoop *) *ml*) C++

Registers the main loop object *ml* with the runtime kernel.

6.4.3 Publishing

To enable users of your module find the exported objects, you may register the string binding handle of the object or objects, along with their type IDs, in any name service or registry that is convenient for you. In releases 1.6--2.0 of ILU, we support an experimental simple binding method that allows you to ‘‘publish’’ an object, which registers it in a domain-wide registry, and then to withdraw the object, if necessary. Potential clients can find the string binding handle of the object by calling a lookup function. **Note that this interface and service is experimental, and may be supported differently in future releases of the ILU system.**

ilu_Boolean **iluObject::ILUPublish** () C++

A method on instances of `class iluObject`, it registers the instance with some domain-wide registration service. The object is known by its <*Server-ID*, *Instance-Handle*> pair. Clients may find the object by passing this pair to the `iluObject::Lookup` function. Returns true if the object can be successfully published in the local registry.

ilu_Boolean **iluObject::ILUWithdraw** () C++

Returns true if the object's registration in the local registry can be successfully withdrawn, or does not exist.

ilu_Boolean **iluObject::ILUPing** (ilu_Error* p_error) C++

Returns ilu_TRUE if the true object exists, and the process serving it can be contacted, otherwise ilu_FALSE. *p_error is set if some error occurred.

6.5 ILU API for C++

6.6 Generating ILU stubs for C++

To generate C++ stubs from an ISL file, you use the program c++-stubber. Three files are generated from the ‘`.isl`’ file (the extension `cpp` is used instead of `cc` when running on Windows):

- ‘*interface-name*`.hh`’ contains the class definitions for the types and procedures defined by the interface;
- ‘*interface-name*`.cc`’ contains the client-side and general code for the interface; and
- ‘*interface-name*`-server-stubs.cc`’ contains the server-side stubs and code for the interface.

Typically, clients of a module never have a need for the ‘*interface-name*`-server-stubs.cc`’ file.

```
% c++-stubber foo.isl
header file interface foo to ./foo.hh...
code for interface foo to ./foo.cc...
```

```
code for server stubs of interface foo to ./foo-server-stubs.cc...
%
```

The option `-renames` *renames-filename* may be used with `c++-stubber` to specify particular C++ names for ISL types. See the following section for more details.

6.6.1 Tailoring C++ Names

It is sometimes necessary to have the C++ names of an ILU interface match some other naming scheme. A mechanism is provided to allow the programmer to specify the names of C++ language artifacts directly, and thus override the automatic ISL to C++ name mappings.

To do this, you place a set of synonyms for ISL names in a *renames-file*, and invoke the `c++-stubber` program with the switch `-renames`, specifying the name of the renames-file. The lines in the file are of the form

construct ISL-name C++-name

where *construct* is one of `method`, `exception`, `type`, `interface`, or `constant`; *ISL-name* is the name of the construct, expressed either as the simple name, for interface names, the concatenation *interface-name*`.`*construct-name* for exceptions, types, and constants, or *interface-name*`.`*type-name*`.`*method-name* for methods; and *C++-name* is the name the construct should have in the generated C++ code. For example:

```
# change "Foo_T_R1" to plain "R1"
type Foo.R1 R1
# change name of method "M1" to "Method1"
method Foo.O1.M1 Method1
```

Lines beginning with the 'hash' character '`#`' are treated as comment lines, and ignored, in the renames-file.

This feature of the `c++-stubber` should be used as little and as carefully as possible, as it can cause confusion for readers of the ISL interface, in trying to follow the C++ code. It can also create name conflicts between different modules, unless names are carefully chosen.

6.7 Threading

The ILU C++ runtime is prepared to be used in either a single-thread or a multi-threaded mode. Single-threaded is the default. To run multi-threaded, the application is responsible for picking a threading mechanism and making it accessible to ILU. Two calls must be made to make a threading mechanism available to ILU. One, `iluServer::SetFork`, enables ILU to fork new threads. The other, `ilu_SetLockTech` (from '*ILUSRC*`/runtime/kernel/iluxport.h`'), supplies various thread synchronization primitives. These calls must be done at startup time, before any ILU servers are created or surrogate objects imported.

6.8 Other ILU Considerations For C++

6.8.1 Libraries and Linking

For clients of an ILU module, it is only necessary to link with the '*interface-name*`.o`' file compiled from the '*interface-name*`.cc`' file generated for the interface or interfaces being used, and with the two libraries '*ILUHOME*`/lib/libilu-c++.a`' and '*ILUHOME*`/lib/libilu.a`' (in this order, as '`libilu-c++.a`' uses functions in '`libilu.a`').

For implementors of true classes, or servers, the code for the server-side stubs, in the file '*interface-name*`-server-stubs.o`', compiled from '*interface-name*`-server-stubs.cc`', should be included along with the other files and libraries.

6.8.2 Initialization order

ILU uses the static-object-with-constructor trick to effect per-compilation-unit startup code. In certain cases you'll want to ensure that a certain compilation unit's initialization is run before another's. While C++ defines no standard way to do this, most compilers work like this: compilation units are initialized (static object construtors run) in the order in which they are given to the link-editor. We want to hear about any exceptions to this rule.

6.8.3 Makefiles

ILU uses the `imake` system from X11R? to produce '`Makefile`'s from '`Imakefile`'s. For more details on this process, Section 19.3.1 [Using Imake with ILU], page 272.

7 Using ILU with Java

7.1 Introduction

This document is for the Java programmer who wishes to use ILU. The following sections will show how ILU is mapped into Java constructs and how both Java clients and servers are programmed. See also `ftp://ftp.parc.xerox.com/pub/ilu/misc/ilu4java.html`

7.1.1 ILU and CORBA

ILU for Java is mostly CORBA compatible.

By this we mean the normal definition of compatibility of english language or of computer science. (In particular we don't use any other organizations definition of "compatible"):

Ilu's compatibility requires the programmers help: Not all programs will be compatible, but it is possible to write compatible programs.

1) Ilu can interoperate with other corba compatible orbs.

2) Applications which are written to use only corba compatible features can switch between ilu and other orbs without changing source code.

ILU for Java's compatibility is not complete or perfect. Not all CORBA orb functions are implemented (DII etc). Ilu implements a subset of the CORBA functionality (for any version), as well as plugs itself into an abstract ORB support interface (if you have a compatible orb, e.g. jdk 1.2 comes with orb delegation packages which ARE compatible).

Ilu generated stubs are not implementing all internal functionality required by other orbs. Stubs generated by other orbs do not provide all internal functionality required by Ilu. This is normally not a problem for any application; should there be a problem, the Ilu stubber has a -prefix option to generate stubs into a different package.

javasoft's interpretation of the standard mapping: (may or may not be accurate): `http://java.sun.com/products/jd` `java.html` The standard mapping is available from OMG as PDF document: `ftp://ftp.omg.org/pub/docs/orbos/97` `03-01.pdf` or a newer version`ftp://www.omg.org/pub/docs/formal/98-02-29.pdf`

7.2 The ISL Mapping to Java

The mapping is similar to the corba mapping. Applications programmers should not notice a difference whether their application use corba stubs or ILU stubs, as long as the application stays withing the corba subset. Many ilu types however implement the java.io.Serializable interface.

7.2.1 Names

In general ISL names are mapped to Java names and identifiers with no change.

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

This mapping, matches the CORBA standard. ILU considers all use of java reserved words, as well as method names of class java.lang.Object to be colliding.

Furthermore, names which have prefixes or postfixes which COULD match identifiers otherwise generated by ILU are considered colliding. (Like "Helper", "Holder")

7.2.2 Interfaces

An ISL `interface` is mapped to a Java package with the same name. For details see the CORBA standard for modules.

Avoid IDL features which are not in the scope of a idl module (= ISL Interface).

The ISL directive (DIRECTIVE-EXPERIMENTAL) options have a few special keys for the java stubber.

Directives which starts with "JAVA-PREFIX" contains a quoted string which is used as a package prefix. This can also be achieved with a stubber command line -prefix option.

Directives which starts with "JAVA-CUSTOMFILE" contains a quoted string which is used as a file name describing custom mapping. This can also be achieved with a stubber command line -custom option.

Directives which starts with "JAVA-REMOTE" contains a list of quoted string which are names of ilu object types (or "*"). Those object types will extend java.rmi.Remote and all methods will declare to throw java.rmi.RemoteException. However, if you combine RMI remotes and ilu objects, you need to know what you are doing or you'll be surprized. (This automaticly implies "JAVA-NOCORBAOBJECT")

Directives which starts with "JAVA-NOCORBAOBJECT" contains a list of quoted string which are names of ilu object types (or "*"). Those object types will NOT extend org.omg.CORBA.Object. Don't use corba specific methods on those (You'd get a type error).

7.2.3 Basic Types

BOOLEAN is mapped in to java boolean.

BYTE is mapped int java byte. ILU carries the bit pattern, not the numerical value.

CHAR and SHORT CHAR are mapped int java char. Values outside their range throw an exception when marshalling.

SHORT INTEGER, SHORT CARDINAL maps into java short.

INTEGER, CARDINAL map into java int.

LONG INTEGER, LONG CARDINAL map into java long.

The unsigned integer types map as bit pattern.

SHORT REAL maps into java float.

REAL maps into java double.

128 bit floating point values are carried around as `"xerox.ilu.types.float128"` but no operations are available.

The `"new"` extended IDL types fixed is not yet supported.

This mapping matches the standard corba mapping.

7.2.3.1 Constant

See the corba standard.

7.2.4 Strings

See the corba standard.

7.2.5 Pickles and Typecodes

See the corba standard for ANY and Typecodes.

At the currenbt time not yet implementing java.io.Serializable

7.2.6 Constructed Types

7.2.6.1 About Java Object serialization

All (*) Constructed Types are mapped into java types which implement java.io.Serializable. Whether the type realy is serializable depends on whether its element type is. Serialization of a constructed type whose element type is not serializable throws an exception.

(*) Two exceptions: Custom surrogates are only serializable if the surrogate type specified so.

Likewise, true object type implementations are only serializable if the implementing class specified so. The implementing class does NOT need to implement any methods, but it needs to implement java.io.Serializable

7.2.6.2 Enumeration

See the corba standard.

Serialization in an ILU context works as expected. In PRE-JDK1.2 however, serialization in a non-ILU context might loose the object identity. (Otherwise equal elements can be detected with `"=="`)

7.2.6.3 Array

Mapped into a java array of the mapped element type. See the corba standard.

7.2.6.4 Sequence

Mapped into a java array of the mapped element type. See the corba standard.

7.2.6.5 Record

See the corba standard.

7.2.6.6 Union

See the corba standard.

7.2.6.7 Optional

ISL Optionals have no corresponding corba type. If the base type of the optional maps into a Java primitive type we will represent the optional with the corresponding Java wrapper or container class. (Boolean, Character, Double, Float, Integer, Short, Byte, or Long). The null value represents absence of the parameter.

If the base type of the optional maps into a Java object type, the optional is mapped as if it weren't optional. The null value represents absence of the parameter.

In the case of Java Strings the difference between a null String and an empty string serves this distinction perfectly.

7.2.7 Object Types

See the CORBA standard for `"`corba interfaces`"`.

7.2.7.1 Surrogate and True Object Types

To request creation of an object the `createTrueObject` *method is called. The implementation of* `createTrueObject` *is expected to call* `super.returnTrueObject` *to register the created object.*

True object have to implement all the interfaces required by their type and by type org.omg.CORBA.Object (unless the Ilu type explicitely doesn't extend org.omg.CORBA.Object)

Unlinke, CORBA, this is all which is absolutely necessary. However, unless there is a need to do otherwise we strongly recommend extending the base class xerox.ilu.IluObjectBase. This will take care automatically of a long list of problems (See the section about garbage collection). But, if you can't use one of those base classes, in ilu you do have alternatives.

If you can't extend a predefined base class, you can make a class of your own and implement the necessary interface. With the ilu distribution is a `"template"` ObjectImpl.template which cut be pasted into your own source code to do so automatically. *See the ObjectImpl.template file in the runtime/java directory.*

7.2.7.2 Methods, Parameters, and Exceptions

7.2.7.3 Java Garbage Collection, Distributed Garbage Collection, and/or COLLECTIBLE

The Garbage Collection features in Ilu look much more difficult then they are. The difficulties arise in supporting various important cases in wild combinations. Most likely your application will not use the whole span of complexities but simply chooses one of the cases.

For Ilu COLLECTIBLE objects (see generic Ilu documentation) do not hang onto objects longer then you need. That is true for both surrogate and true objects.

Non COLLECTIBLE objects are alive until deleted. If you are providing a true object implemementation, it is your responsibility to prevent this object from being garbage collected; Ilu does not necessarily do so. If the program lets a true object being garbage collected that means Ilu will eventually stop serving it.

There are several means to have Ilu help keeping an object alive:

If possible, true objects should extend IluObjectBase (or use the template). This would take care of all the complexities following.

Most direct, the xerox.Ilu.registerTrueObject method provides a lifetime argument. This should be a lifetime value as is provided by the xerox.ilu.IluLifetimeArgs class. However since many registrations happen implicitely or in places which do not provide that argument there are other means to specify lifetime.

There are two marker interfaces xerox.Ilu.IluLifetimeForget and xerox.Ilu.IluLifetimeRemember which can be implemented by true objects. Doing so guides Ilu's behaviour if xerox.Ilu.registerTrueObject does not specify an explicit method or xerox.Ilu.registerTrueObject is not explicitely used at all.

Of course the application could simply manage the lifetime itself, but see the note below.

Note that there is a big difference in whether a true object is extending IluObjectBase or not. Ilu goes to tremedous troubles to not require applications to extend a predefined base class. However if this service is not needed we recommend applications to simply extend IluObjectBase for true object types. By doing so, Ilu can take advantage of inherited code which helps it handle lifetime. If Ilu can finds no hint at all that the application is doing something about the lifetime, Ilu's behaviour might become more conservative then the programmer likes. (For CORBA users: using org.omg.CORBA.portable.ObjectImpl.java works as well as xerox.ilu.IluObjectBase, but you can only do corba things with it. Use xerox.ilu.IluObjectBase otherwise.)

Note to the curious why the internal complexity is necessary: Ilu is implementing weak refs, even on jdk version which have no support for it.

ILU Java reads the `ILU_COLLECTABLE_CINFO` environment variable for setting up ports for the special object used to guide collectible objecs.

7.2.7.4 Java serialization

Surrogates implement java.io.Serializable. We recommend to our implementors to make true object types serializable as well. The IluObjectBase interface does so.

7.2.8 How to make your idl Java friendly

Note that you don't have to do this. Ilu will mangle your names if necessary.

However, if you keep your names Java friendly, your stubs will be easier to read and debug by people used to popular java coding standards or the CORBA standard mapping. If you avoid name conflicts no esoteric name mangling needs to happen and your ilu stubs are more likely to match the CORBA standard correctly.

1. Avoid Java keywords. (e.g. boolean, int, class, for, ...)
2. Avoid names of Java standard classes (in the java.lang package). (e.g. String, Integer, RuntimeException)
3. Avoid suffixes which are used by CORBA standard for your type and class names. (e.g. Stubs, Operations, Helper, Holder, ImplBase)
4. Avoid prefixes which are used by CORBA standard for your type and class names. (e.g. POA_)
5. Avoid suffixes which are generated by ILU for your type and class names. (e.g. _exh_)
6. Avoid gratuidous typedefs. (But it is ok to use typedefs for functional reasons)
7. Avoid using the same name for the interface and a type.
8. As a courtesy, try to also avoid names which cause problems for other programming languages.
9. Start your types, classes and exceptions with capital letters.
10. Start your methods and enumeration values with small letters

7.3 Access to standard ILU features

7.3.1 Servers and Ports

7.3.1.1 True Servers

Each object exported by an implementation must belong to a true server, an instance of the Java type `IluServer` which is implemented by the `xerox.ilu.IluServer` class.

An `IluServer` instance can be created by calling the function `ilu.createServer(`*[serverID]*`)`, which returns a value of type `IluServer`. If *serverID* is a string, it specifies the server ID; if it is the Java value `None`, a server ID will be invented automatically.

Other methods allow the specification of a daemon flags or ports for different transports or protocols, or an *object table*, which allows specification of a callback function for creating true instances on demand.

The first time a true server is created, it becomes the default server. The default server is used for an exported object if a server is not otherwise specified or when it is explicitly requested with the `defaultServer` method. If an object is exported before any servers have been created, one will be created automatically using default parameters and possibly a message to that effect will be written to `System.err`.

An object of type `IluServer` has an accessor method `serverId()` that returns its server ID.

Look at the Java classes IluServer and IluPort for details. In general servers may be omitted and a default server is used. IluPort as well may be ommited.

ILU for Java reads the `ILU_DEFAULT_CINFO` environment variable for setting up default ports.

See the section Java Garbage Collection, Distributed Garbage Collection, and/or COLLECTIBLE for how to setting up the port used for implementing COLLECTIBLE objects.

7.3.1.2 Object Tables

Object tables are defined in the class IluObjectTable. To define an object table one subclasses `IluObjectTable` and attached the subclasses to an `IluServer` when the server is created. An `IluObjectTable` can only be attched to one `IluServer`.

Because the Object Tables calls are made with internal ILU locks held, it must not do recursive calls into ILU.

7.3.2 Threading and Event Loops

ILU uses Java threads. There is no event loop option.

It is possible to change thread priorities by setting properties at start up of the application. This is however *not* recommended.

A few arguments can be passed by setting "properties" on the thread. This is motivated by the java standard mapping which has no place to pass extra arguments.

However Java threads, if not subclassed do not support "properties". To set thread properties, calls need be performed from an xerox.ilu.IluServiceThread

7.3.3 Custom Records

Record types are mapped to Java objects. It is possible provide a subclass which will be used by ILU instead.

"Custom Records" class assign their own class to a (protected) java.lang.Class variable (_theClass) of the original stub class for the record. Whenever ILU needs to allocate an instance of such a record it will allocate a an instance of the subclass instead.

7.3.4 Custom Surrogates

Surrogate creation can be intercepted. To do so one registers either a Java class, or, a factory object with the stub class implementing an object type. See the methods `IluClassRep.setSurrClass` or `IluClassRep.setSurrFactory` for details.

7.3.5 String Binding Handle Formation

The class `xerox.ilu.IluSBH` provides methods for both scanning and composing string binding handles.

7.3.6 Simple Binding

An object may be published using the simple binding service by calling the method `xerox.ilu.IluSimpleBinding.publish()`.

An object may be unpublished by calling the method `xerox.ilu.IluSimpleBinding.withdraw()`.

A published ILU object may be obtained by calling `xerox.ilu.IluSimpleBinding.lookup(`*sid*`,` *ih*`,` *cl*`)`, where *sid* is object's server's server ID, *ih* is the object's instance handle, and *cl* is its class.

As alternative to IluSimpleBinding, the `resolve_initial_references` method from org.omg.CORBA.ORB could be used. See another alternative in the chapter about CosNaming.

7.3.7 Principal Identities and Passports

Passport are represented in the class `IluPassport`. Identities are represented by the class `IluIdentity`. The class `IluIdentityType` represents the type of an identity. A passport can carry at most one IluIdentity of each `IluIdentityType`.

Client generated passports are mutable and stay alive until garbage collected. Clients tell ILU about their passports via the current thread which must be a subclass of IluServiceThread. The methods `IluServiceThread.setMyStubPassport` and `IluServiceThread.setStubPassport` can be used to set passports. For security reasons passports of other threads can not be set. (We need to rethink the possible threats). However it is possible to clear the passport of any thread (as this is only a denial of service attack).

ILU generated passports are immutable and can be used only while the call for which it has been constructed is ongoing. If used afterwards, an exception will be raised. For a client to access an Ilu generated passports, the method `IluPassport.getSkeletonPassport` is used.

These passport features are carried around fully, but since the api to create identities and identity type features are still missing, usefullness of passports is grossly limited.

7.4 Building ILU-for-Java Applications

7.4.1 Stub Generation

Please don't confuse the java stubber (called javah) with the ilu stubber (called java-stubber). javah is used to create native method interfaces. java-stubber is named that way to provide a naming scheme matching other ilu stubbers for other programing languages.

To generate Java stubs from an ISL file, use the program `java-stubber`. Since the stubber creates a surprisingly large number of Java files, these files must be directed to a designated stub directory.

Using a designated stub directory is only a default and could be changed. We strongly advice against changing this to protect your manually written source files from being overwritten.

To be able to compile the huge list of files, the stubber generates a file with a list of generated Java files is generated.

The stubber recognizes the following options or switches:

- **-dir** *DIRECTORY PATH*
  The stubs are generated into the directory specified by the *DIRECTORY PATH* argument. This command line option could direct the stubs into the working directory, but if omitted the default path is '`./javastubs`'.
- **-xrx**
  The old Xerox PARC Java/ILU mapping is used for this interface. (See also **-omg**.) The main difference between **-xrx** and **-omg** option are the holder classes; we are working on eliminating this difference.
- **-omg**
  More or less what we imagine is OMG's mapping for this interface is used. Remember that this is about the interface to the application, not about the interface to the orb.
- **-flat**
  Generate all stubs into the designated stub directory directly. This is not recommended because files can be overwritten. The default behavior is to write stubs in subdirectories according to their interface names.
- **-prefix** *PACKAGE_PREFIX*
  Generates stubs into a package prefixed with the named *PACKAGE_PREFIX*.
- **-noprefix**
  Generates stubs without package prefix. Recommended for serious interfaces; NOT recommended for experimenting.
- **-noserv**
  Generates stubs which save a few bytes but can't be used on the true side. NOT recommended.
- **-reportfile** *REPORT-FILENAME*
  Additional parameter for a file name. A list of generated Java files is generated. This list can be used to make makefiles, etc. **-reportfile** has no 'suffix' version (see below).
- **-noreport**
  The list of generated Java files is *not* generated. **-noreport** has no 'suffix' version (see below).

- **-exsuffix**
  Define a suffix used in the name of exceptions. NOT recommended.

All command line options (unless if documented otherwise) have a second variant with a suffix `"1"`. If the command line option with suffix is used, an extra argument for the name of an interface is read, and the option only affects that interface. The un-suffixed command line options affect all interfaces. Suffixed command line options override un-suffixed command line options. Option processing is from left to right.

The stubber can be run without arguments to generate a usage message.

7.4.1.1 Problems and Solutions

- **Stubbing will create a very large number of files with the Java extension. You don't want to loose your source files among the generated stubs.**
  We recommend creating a subdirectory called '`./javastubs/`'. All generated stubs should be generated into this directory. (This is the default; we do not recommend changes.) *(As a safety measure) if a non-standard directory is used, the stubber does NOT create it automatically. Unlike the top level stub directory, sub directories are created automatically by the stubber.*
- **Stubs might have conflicting names.**
  Generate stubs in a hierarchical directory structure, according to their Java package structure. (This is the default; we do not recommend changes.)
- **Stubs might conflict with manually generated Java sources.**

  1. Use good ILU interface names (Or OMG IDL module names). This is recommended for ''production'' ILU use.
  2. Direct the stubber to generate all stubs into a prefix package stubs. This is recommended for perusing non-well-known interfaces. Use the **-prefix** option to name a prefix.
  3. **Stubs conflict with standard stubs loaded with the bootclass path.** Use the -prefix option when stubbing.

7.4.1.2 Java Compilation

Unless you are using make files, we recommend stubbing into empty directories. This allows the Java compiler to compile all Java files with an asterisk syntax. Always use the **-d** option with the Java compiler!!!

In unix, the report file can be used to generate the list of files to be compiled. It can be used directly with a back-quote cat report-file back-quote syntax.

7.4.1.3 Execution

At execution time the loader needs to find

1. **Your application and its regular imports (binaries; resp. classes).** This is normal Java usage. It might be easiest to the `CLASSPATH` environment variable.
2. **Java binaries (classes) from generated stubs.** There is nothing special here. Compile the stubs into whatever directory you want (e.g. '`./classes/`') and put that directory on the `CLASSPATH` environment variable.
3. **Java binaries (classes) from ILU runtime support.** The classes binaries from the ILU runtime support are compiled into the jar file '*ILUHOME*`/lib/ilu.jar`'; put this file on your `CLASSPATH` environment variable. Also put the jar file '*ILUHOME*`/lib/ilujavaobv.jar`' on your `CLASSPATH` environment variable if you use Java objects by value or Java marshalled objects.

   (Ilu's files from the org.omg class conflict with corresponding implementations deliverd by Sun microsystems. Ilu run's with either versions, your choice in jdk1.1. However in jdk1.2, because of Sun's version are bootclasses, replacing them might be difficult.)

   Java JDK 1.1.4 starts to garbage collect classes. There was a known bug in JDK about collection of classes (fixed in jdk1.1.6). ILU depends on its classes not being garbage collected and has some strange code to prevent such collection even in case of the buggy jdk's. Nevertheless it might be conservative to use the **-noclassgc** switch with jdk1.1.4 and jdk 1.1.5.

   If your java system does not support jar files, the classes are compiled into '*ILUHOME*`/lib/classes/`'.
4. **C binaries (object files) from ILU runtime support and kernel.** The C binaries (object files) are accessed using the `LD_LIBRARY_PATH`. The path must have an entry pointing into the ILU Java runtime directory.

7.4.1.4 Debugging

Use of the standard ILU debugging environment variable `ILU_DEBUG` is recommended; please Section 16.7 [Debugging ILU Programs], page 236.

Java ILU listens to a large number of Java command line switches. The simplest one is to set the **-D** command line option (e.g. **-Dilu.debug=4**). For more specialized command line switches look at the `xerox.ilu.IluDebug.java` file. Looking at the well documented source code is more relieble then duplicating the rather fast changing flags in the documentation.

Note that the **-D** option of the Java interpreter has no relationship whatsoever to the **-d** option of the Java compiler.

When using `dbx` or `gdb`: The standard Java binaries (classes) are found automaticly (without specification on the `CLASSPATH` environment variable) when not debugging. When debugging, those classes *must* be specified on the `CLASSPATH`. We don't understand this difference.

7.4.1.5 Debugging native ILU code

When setting break points with dbx or gdb it is necessary to make ilu wait first. Ilu supports the -Dilu.interactivewait=true property to add an extra interactive wait. For details look at the class xerox.ilu.IluInit2.

7.4.2 Implementing an ILU module in Java

A Java program which wants to implement a object type *T* from interface `I` needs to create a class which implements the stubber-generated Java interface `I` and java class org.omg.Object (Preferably by extending the xerox.ilu.IluObjectBase class). Objects of this class then are made available (registered) with the ILU runtime either implicitly or explicitly, or with an object table.

Explicit registration allows specification of additional parameters, like the string binding handle, the server, and more. Explicit registration done by calling the stubber generated method `I.`*T*`_stub.registerTrueObject()`. As an alternative, if the programmer is willing to specify the class, he can also use the method `xerox.ilu.Ilu.registerTrueObject` directly.

Objects are implicitly registered with ILU if they are returned as a result value from a method call that returns an object type or has an object type as an INOUT or OUT parameter or if they are published using the Simple Binding mechanism.

7.4.3 Hints about implicit registration

If objects are not registered explicitely but get auto-registered on use, there are two pitfalls to watch out for.

1. The ILU type must be unique. If no type is given ILU will look at the Java type and make its best possible guess about the ILU type.
2. The ILU type must be loaded. Unless the ILU type is loaded the registrar will never find that type. If there is any doubt about whether a stub implementing an ILU type is loaded or not, it might be useful to actually load the class. The ilu.load property can be used to load a class from external commands. The stubber also generates a special class whose sole purpose is to help load whatever is necessary for the registrar to not miss a class.

7.4.3.1 Java's single inheritance classes.

If you can not extending the xerox.ilu.IluObjectBase class, there is a template ObjectImpl.template which you can cut and paste into the source code of your own class.

Alternatively you can use what other orbs call the tie approach. Check the stubber options on how to generate tie classes. When using the tie approach you should find a means to give your tie class an appropriate life time.

A client program may obtain a reference to an ILU object in one of the following ways:

1. **True instance creation**.
   The application may create an true instance of a class.
2. `objectFromSBH()`
   Knowing the string binding handle *sbh* and class (`xerox.ilu.IluClassRep`) (or superclass) *cl* of an object, call `xerox.ilu.Ilu.objectFromSBH(`*sbh*`, `*cl*`)` which returns an instance of that class.

   For example, to obtain an instance of ISL type `square` from `INTERFACE shapes`, which has a string binding handle *sbh*, one would call `xerox.ilu.Ilu.objectFromSBH(`*sbh*`, shapes.squareStub.iluClass())`.
3. **ILU Simple Binding**.
   Knowing the object ID *(sid, ih)* and class *cl* of an object that has been published using the simple binding service, call `xerox.ilu.IluSimpleBinding.lookup(`*sid*`, `*ih*`, `*cl*`)` which returns an instance of that class (or raises an exception if the lookup fails).
4. **Return result or parameter**.
   An instance may be received as a result value from a method call that returns an object type or has an object type as an `INOUT` or `OUT` parameter.

To get the ilu class (java type `xerox.ilu.Ilu.IluClassRep`), use one of these methods:

1. The static iluClass() method generated with the stub class.
2. The static xerox.ilu.IluClassRep.fromIluObject method with an object instance argument.
3. The xerox.ilu.IluClassRep.fromIluClassName method with the ISL name of the class as argument. For Ilu to find the class it is necessary that the java stub implementing the ISL class has been loaded.

7.5 Notes for ILU for Java in applets

Use of ILU in applets is not impossible, but it is not fully supported either. Read about the problems to see what you can do and what you can't.

The highlevel problem are native code and no support for class unloading.

ILU has native methods. That means Ilu must be loaded somewhere/somehow where native methods are supported. ILU has been loaded into Hotjava, java-plugin (Activator). This can be solved in JDK 1.1 by loading ILU on the classpath, or by using signed applets (which has not been tested).

ILU also has been loaded into the regular Netscape 4.05 VM, but that needs special setup in the Ilu build process. (See Imakefile and source/runtime/java/IluMozillaExtra.java). This has not been tested in the recent release and is depricated in favor of plugin.

In JDK1.2 Ilu can be made an installed extension. (Downloaded extension in JDK1.2 don't yet support native methods, even when signed). If you run Ilu as an extension you also need the stubs in the extension, otherwise the classloader would throw a security exception.

Likewise, if you manage to put Ilu into the bootclasses you also need the stubs in the bootclasses, for the same reason.

To run Ilu as an extenson, out the jar files in the proper extension directory, but don't forget to put the native code into the right place also.

ILU does not support class unloading. ILU itself is quite security aware, but there are still some problems. In particular, there is a security problem if an applet loads malicious stubs and later another applet things these are correct stubs. To prevent class unloading, stubs should be in the same classloader as the ILU runtime: Either both signed and dynamic, or both on the classpath, or, both as an extension.

Very little testing of running ILU in a browser or a java-pluggin has been done. Set up your environment that stub classes will never be unloaded.

7.6 Notes for Java on particular architectures

7.6.1 Notes for Java on Windows

Please see the "Java" subsection of "Building ILU" in the chapter Section 10.9 [Using ILU with Microsoft Windows], page 183.

7.6.2 Notes for Java on Linux

Users of Linux Redhat 5.1 reported problems when Ilu was configured using the "poll" system call. The configuration option --with-java-nopoll can be used to force use of "select" instead "poll".

7.6.3 Native threads versus green threads

On UNIX systems, Sun's Java runtime exports one or both of two different interfaces to its multi-threading support, depending on which of two implementations is used. The choice is between so-called "green threads", which are "user level" threads entirely implemented in the Java runtime, and "native threads", which use the thread support offered by the OS and/or C runtime.

On Win32 systems, only native threads are available. In case that the ILU for Java system is built with JNI, the difference between "green threads", and "native threads" does not matter.

ILU faces the choice between native and green threads at two times. The first is when ILU is built: ILU can be built for use with either or both kinds of Java threading system. ILU's configuration script (used at ILU build time) tries to figure out which kind(s) of Java threads are available in the local Java implementation and build ILU support for as many kind(s) of Java threads as are available. If it gets this wrong, you can explicitly tell it which kind(s) of threads to build support for, with a configuration option: `--with-java-threads=`*KINDS*, where *KINDS* is one of `green`, `native`, or `both`.

The second time when ILU must choose between Java threading systems is at loading time for Java native code library. Sadly we do not know how to determine whether Java is using green threads or native threads. ILU simply tries one version, and, if that throws an exception and another version is available, tries the other version. (The order is important: on a certain Java VM and threading model the wrong native code didn't throw exceptions but caused a crash.)

7.6.4 JNI versus ONI vesus RNI

Ilu is configured differently according to what native method interface is used by your java VM. Mostly, the configuration step will choose the right system for you.

The choice of RNI is quite obvious: Use RNI on microsoft sdk and nowhere else. (This happens automatically)

The choice of JNI versus ONI is more difficult.

In jdk1.1 we recommend using ONI because JNI doesn't implement weak pointers, the java VM's don't do compaction, and, ONI has been used in most testing.

In jdk1.2 we recommend using ONI IF your system supports ONI and does NOT use memory compaction. Otherwise we recommend using JNI.

The ILU configuartion process chooses automaticly whether to use JNI or ONI. Should this choice be wrong, it can be overridden with the `--with-java-jni` option.

7.6.5 Java releases

The ilu build builds one version of binary ilu. However, the filenames for the binaries are distinct. It is possible to do one ilu build for jdk1.1 and one ilu build for jdk1.2... each in the same directory. This works because the by accident or design, the java classes for jdk 1.1 and jdk1.2 are compatible. Ilu at runtime decides which version of the native libraries to load.

Ilu used to work with various jdk1.2beta releases but after usage of jdk1.2 Ilu went through major changes in the configuration and beta releases have thereafter never been tested anymore.

7.7 CosNaming

Sadly, Ilu stubs and standard stubs are not interchangable. In Java jdk1.2 the org.omg.CosNaming stubs are in the Java core and Ilu has no chance of substituting its own stubs. There are two possible ways to handle this problem:

a) This is for people who want to use Ilu strictly corba compatible. Stub the CosNaming.idl with the standard org.omg prefix. Load a version of java which does not have CosNaming from another source. If an application can do this its use of CosNaming is corba compatible in source code. (It is always corba compatible on the wire).

b) If your application can not replace the standard CosNaming packages. Stub the CosNaming.idl with another prefix, for example xorg.omg. The stubs will have a different package name, but they will be corba compatible on the wire.

We recommend using CosNaming.idl from ilu's etc/CosNaming directory. The reason for this is that the idl does not have a package prefix in the module name but only as a pragma. So whatever prefix is generated when stubbing CosNaming.idl, the names used on the wire get picked from the pragma and your CosNaming stubs look right on the wire.

7.7.1 CosNaming Naming Server

Should you use ILU's CosNaming Naming Server? It really doesn't matter. The service is wire compatible and accessed via your stubs. Ilu's service and somebody elses service are interchangeable.

7.7.2 Bootstrapping with CosNaming

It is possible to start up Ilu like a regular orb and bootstrap via the CosNaming service. There is an example in the chapter about the ORB class. See also `ftp://ftp.parc.xerox.com/pub/ilu/misc/iluSunJavaIDL.html`

You will need to specify that ILU is used as the ORB. To do this use a property like this: -Dorg.omg.CORBA.ORBClass=xerox.ilu.IluORB.

You will need to force your application to load the required stubs. use a property like this: -Dilu.load=xorg.omg.CosNaming._allJavaStubs:HelloApp._allJavaStubs This usage demonstrates loading both, the stubs from CosNaming as well as the stubs from the application.

You will need to tell ILU what the string binding handle of the root context of the naming service is. One way is to copy the IOR from when the naming service was started, as a propert like this: -DNameService=IOR:0000000000000028494 44c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f6e7465787

The other way is to write the IOR into a file or an URL. In unix this could even be done by redirecting the output when starting the naming service. Use a property like this to read the IOR from a file: -DNameService=file:/tmp/file.temp or like this to read a URL: -DNameService=url:http://x.y.com/foo.htm

You might use the xerox.ilu.IluORB.resolve_initial_sbh or xerox.ilu.IluORB.readSBHFromURL methods directly to make up your own way of bootstrapping Ilu.

7.8 ILU for Java Reference

The interface is rather large. This section will cover the most important classes but reading the standard javadoc documentation or reading the source will be more complete.

7.8.1 javadoc documentation

ILU java code contains javadoc tags. We think however that reading the source code might be easier then reading the javadoc output.

javadoc documentation: `http://java.sun.com/products/jdk/javadoc/index.html` The ILU build does not automaticaly generate the javadoc documentation.

7.8.1.1 Build of the javadoc documentation

The ILU build does not automaticaly generate the javadoc documentation.

With jdk1.1 you need to manually create an images subdirectory and copy the images. (This has been fixed in jdk1.2). Since I don't know how to make the ilu build find the images, the build does not do the javadoc step automatically.

Another reason to not automatically build the javadoc documentation is that different users might want to or not include -package and -private features. The third reason for manual build is that some users might want to include standard packages into the same documentation build. For some java release the way to make javadoc comments is this:

```
# create a javadocs directory
# we recommend using ILUHOME/doc/javadocs
# (When making links, they might point to that directory)
mkdir javadocs

# Go to the java sources
cd ILUSRC/runtime/java/jsrc

# build the javadoc documentation into the javadocs directory
javadoc -d javadocs xerox.basics xerox.ilu xerox.ilu.tools org.omg.CORBA
org.omg.CORBA.portable org.omg.PortableServer

# copy the javadoc images into the proper place (not necessary with jdk-
1.2)
# (Substitute the source as you most likely have the images somewhere else...)
cd javadocs; mkdir images; cd images; cp /project/java/jdk-1.1/docs/api/images/*
.
```

The java runtime source directory contains a script "buildjavadoc" which could be used to create javadoc documentation using JDK1.2

7.8.2 Description of Java classes

7.8.2.1 CORBA Exceptions

Ilu has its own version of CORBA Exceptions, or, uses the CORBA Exceptions coming with the standard distribution. This is a configuration choice.

1. org.omg.CORBA.UserException This is the base class for all generated user exceptions.
2. org.omg.CORBA.SystemException This is the base class for all system exceptions.

1. org.omg.CORBA.BAD_CONTEXT
2. org.omg.CORBA.BAD_INV_ORDER
3. org.omg.CORBA.BAD_OPERATION

4. org.omg.CORBA.BAD_PARAM
5. org.omg.CORBA.BAD_TYPECODE
6. org.omg.CORBA.COMM_FAILURE
7. org.omg.CORBA.DATA_CONVERSION
8. org.omg.CORBA.FREE_MEM
9. org.omg.CORBA.IMP_LIMIT
10. org.omg.CORBA.INITIALIZE
11. org.omg.CORBA.INTERNAL
12. org.omg.CORBA.INTF_REPOS
13. org.omg.CORBA.INV_FLAG
14. org.omg.CORBA.INV_IDENT
15. org.omg.CORBA.INV_OBJREF
16. org.omg.CORBA.MARSHAL
17. org.omg.CORBA.NO_IMPLEMENT
18. org.omg.CORBA.NO_MEMORY
19. org.omg.CORBA.NO_PERMISSION
20. org.omg.CORBA.NO_RESOURCES
21. org.omg.CORBA.NO_RESPONSE
22. org.omg.CORBA.OBJ_ADAPTER
23. org.omg.CORBA.OBJECT_NOT_EXIST
24. org.omg.CORBA.PERSIST_STORE
25. org.omg.CORBA.TRANSACTIONREQUIRED
26. org.omg.CORBA.TRANSACTIONROLLEDBACK
27. org.omg.CORBA.TRANSIENT
28. org.omg.CORBA.UNKNOWN

7.8.2.2 Holder classes

A means to pass a CORBA things as a reference parameter. Holder classes are mutable.

Ilu has its own version of Holder classes, or, uses the CORBA Holder classes coming with the standard distribution. This is a configuration choice.

1. org.omg.CORBA.AnyHolder
2. org.omg.CORBA.BooleanHolder
3. org.omg.CORBA.ByteHolder
4. org.omg.CORBA.CharHolder

5. org.omg.CORBA.DoubleHolder
6. org.omg.CORBA.FloatHolder
7. org.omg.CORBA.IntHolder
8. org.omg.CORBA.LongHolder
9. org.omg.CORBA.ObjectHolder
10. org.omg.CORBA.PrincipalHolder
11. org.omg.CORBA.ShortHolder
12. org.omg.CORBA.StringHolder
13. org.omg.CORBA.TypeCodeHolder

7.8.2.3 CORBA classes

Ilu has its own version of CORBA classes, or, uses the CORBA classes coming with the standard distribution. This is a configuration choice.

Some of Ilu's own version of corba classes do not completely correspond to the standard. The intend is that a the corba classes look and work like the standard classes as seen by client applications SOURCE code.

1. org.omg.CORBA.Any A skeleton class to achieve corba source compatibility
2. org.omg.CORBA.BOA There is no use for this in Ilu. This type is provided to make corba compatible calls...
3. org.omg.CORBA.CompletionStatus
4. org.omg.CORBA.CORBA_ObjectHelper The helper class for CORBA::Object
5. org.omg.CORBA.CORBA_ObjectStub For the implementation of CORBA::Object
6. org.omg.CORBA.portable.ObjectImpl CORBA (but not ilu) requires all true classes to inherit from this class.
7. org.omg.CORBA.Object CORBA::Object. (WARNING While all ilu objects have implement the java interface org.omg.CORBA.Object, that doesn't mean the corresponding ISL or IDL type is a subclass of CORBA::Object! Use runtime typing)
8. org.omg.CORBA.ORB Makes Ilu look like it would be a CORBA ORB

   The way `resolve_initial_references` works:

   The argument is used as a key to access the ORB properties. (The ORB properties can be set either with init, or if not set default to the java system properties and, in case of applications it might read environment variables). The property is accessed; its value is supposed to be an ilu SBH. The object for this SBH is accessed and returned by resolve_initial_references.

   There is an extra twist: If the SBH string starts with `"file:"` that adds an extra level of indirection: Ilu will access the file and use the contents of the first line as the SBH instead. (And even better: If the first line looks like a re-direct from Sun's Java IDL naming service, ilu will read the second line...)

A typical start up using this feature coold look like this:

```
org.omg.CORBA.ORB orb = ...
org.omg.CORBA.Object objRef =
     orb.resolve_initial_references("NameService");
org.omg.CosNaming.NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

When starting the application, the property can be set on the command line, for example:

java -Dorg.omg.CORBA.ORBClass=xerox.ilu.IluORB -Dilu.load=org.omg.CosNaming._allJavaStubs:MyApplication.
-DNameService=IOR:000000000000002849444C3A6F6D672E6F72672F436F734E616D696E672F4E616D696E674
startMyApplication

or maybe

java -Dorg.omg.CORBA.ORBClass=xerox.ilu.IluORB -Dilu.load=org.omg.CosNaming._allJavaStubs:MyApplication.
-DNameService=file:/tilde/yourName/fileName startMyApplication

The above example loads the CosNaming stubs from the command line. The reason for this is to make sure that the stubs are actually loaded before orb.resolve_initial_references is called. Ilu requires types to be known (stubs loaded) before an object of that type is read, otherwise the type information will be lost.

Warning: In jdk1.2 the CosNaming stubs are in the java core and not replaceable by Ilu. This example works only jdk1.1 based, or in jdk1.2 when fooling with the boot class path. In regular jdk1.2 use we recommend changing the package prefix for the CosNaming stubs, maybe use xorg.omg instead org.omg (therefore also update the application and the -Dilu.load properties).

9. org.omg.CORBA.TypeCode
10. org.omg.CORBA.TCKind

7.8.2.4 Basic environment classes

These classes are public, but not necessarily considered part of ilu.

1. xerox.basics.Consumer0 A simple queue with a thread which consumes all elements provided. Class is public; used by ilu but not part of the ilu public api.
2. xerox.basics.Environment A simple means to access or specify the environment. Class is public; used by ilu but not part of the ilu public api.

   The command line flag interface to load additional classes looks usefull to ilu, but MUST NOT be used from ilu client applications directly. Ilu duplicates this functionality with an ilu.load property. The difference is the initialization order! The ilu.load property avoids bad initialization order loops.
3. xerox.basics.IntTab An IntTab is a hash table which associates int keys with objects.
4. xerox.basics.IntTabEnumerator For enumerating IntTab values
5. xerox.basics.Queue A simple queue class.
6. xerox.basics.NowhereStream A trivial OutputStream class
7. xerox.basics.NowhereWriter A trivial Writer class

8. xerox.basics.VMExtras

   Remembers objects; better then using static variables as VMExtras also deals with collecting classes.

   High priority thread introducing proper timeslicing into the Java vm.

7.8.2.5 ILU exceptions

1. org.omg.CORBA.SystemException
2. org.omg.CORBA.UserException

1. xerox.ilu.IluRuntimeException IluRuntimeException is super class of all IluRuntimeExceptions. RuntimeException's don't need catch phrases.
2.

   xerox.ilu.IluCustomMappingException
3. xerox.ilu.IluInconsistentCallException
4. xerox.ilu.IluUnexpectedException

7.8.2.6 ILU public classes

1. xerox.ilu.types.float128 A class for 128 bit floats.
2. xerox.ilu.Ilu The top level public interface to Ilu
3. xerox.ilu.IluAny Ilu's concrete class to holdCORBA::ANY
4. xerox.ilu.IluBatcher An IluBatcher represents a batching scope. Calls made within a batching scope are batched by the ILU runtime.
5. xerox.ilu.IluClassRep This represents an ilu class
6. xerox.ilu.IluCustomMapping An ilu type which supports custom mapping needs to register an object of IluCustomMapping-type to perform the actual transformation.
7. xerox.ilu.IluDataSupport An object implementation can help ILU by remembering internal data.
8. xerox.ilu.IluDebug The api is private, but the the command line flag interface is of interest to ilu users.
9. xerox.ilu.IluEnvironment The api is private.
10. xerox.ilu.IluIdentity Representation for ilu_IdentityInfo
11. xerox.ilu.IluIdentityType Representation for ilu_IdentityType
12. xerox.ilu.IluIHProposer A true object can support this interface to help Ilu in making up an instance handle. Ilu uses this method if an object is registered without providing an actual instance handle.
13. xerox.ilu.IluLifetimeArgs Keys to specify lifetime argumens of corba object
14. xerox.ilu.IluLifetimeForget Interface to specify lifetime argumens of ilu object
15. xerox.ilu.IluLifetimeRemember Interface to specify lifetime argumens of ilu object
16. org.omg.CORBA.Object Generic, client visible interface; super interface for all ilu objects (unless object type explicitly requires non-inheritance) Sadly has a few methods which are useless but must be implemented.

17. xerox.ilu.IluObjectBase Base super class used for surrogate object implementation. Class is client visible. This might as well be used as base class for true objects, at the discretion of the programmer (of the true object's class). In fact it is HIGHLY RECOMMENDED to extend this class if possible.
18. xerox.ilu.IluObjectTable An object table gives the application the ability to create true objects upon presentation of an instance handle. This is used by the application to pass in to the creation of a true server. This must be subclassed to be useful.
19. xerox.ilu.IluPassport Representation for ilu_Passport.
20. xerox.ilu.IluPickle Representation of typed values
21. xerox.ilu.IluPort This is the Java representation of the concept ilu_Port. Server applications should create ports with the IluServer interface. However this interface provides additional methods to destroy ports.
22. xerox.ilu.IluRootObjectHelper Corba style `"Helper"` class for the ilu root object.
23. xerox.ilu.IluRT0 Top level class. However we recommend using xerox.ilu.Ilu for all features available in xerox.ilu.Ilu and want to keep xerox.ilu.IluRT0 easier to be changed.
24. xerox.ilu.IluSerializationContext An IluSerializationContext represents an instance of the serialization guarantee. An instance is with respect to a particular server and set of calls. The guarantee is that the server application code receives calls in the same order as the client application code makes them, except that calls made after a barrier call can start service before calls made before the same barrier call. A barrier call is one that raises a system exception whose completion status is MAYBE (in CORBA terms, which are not quite right to use here) or that raises the barrier error. The client may not issue any two of these calls concurrently. <p> The way for a client to associate an IluSerializationContext is to attach the IluSerializationContext to the thread. This works only from IluServiceThread-s.
25. xerox.ilu.IluSBH String binding handle access. Parsing or composing SBH's takes the arguments and builds and IluSBH structure. The desired information then can be accessed with accessor functions. Currently IluSBH structure are immutable, however you should not rely on this immutability: future releases may or may not make this type more lightweight, mutable, and the fields unprotected.
26. xerox.ilu.IluServantFinalizable A server object can implement this interface to request a call of iluServantFinalize on its real finalization time. Please no resurection or other fancy usage.
27. xerox.ilu.IluServer This is the Java representation of the an ilu_Server. Only true objects have their servers accessible to clients
28. xerox.ilu.IluServerRelocation Interface used for clients which need to initiate server relocates.
29. xerox.ilu.IluServiceThread Subclass for certain ILU threads. This class is used to be able to transmit extra environment or thread specific information between the ILU kernel and its application. Threads forked by ILU for serving skeleton methods do use this thread class. Applications optionally may use this thread class when requiring service from ILU. Application use is optional.
30. xerox.ilu.IluSimpleBinding Simple binding for ILU.
31. xerox.ilu.IluTransportInfo Transport information is useful to describe servers.
32. xerox.ilu.IluTypeCode For representation of types...

33. xerox.ilu.IluTypeKind For representation of types...

7.8.2.7 Internal or esoteric classes

No public client use allowed or expected. Either because class is private for stubs or because it is esoteric in nature. The source of these classes documents why no use is excpected.

1. xerox.ilu.tools.gnh
2. xerox.ilu.tools.japp

1. xerox.ilu.corba_ServantFinalizer
2. xerox.ilu.IluAlarmTech
3. xerox.ilu.IluCall
4. xerox.ilu.IluClassAccess
5. xerox.ilu.IluConstantValueKind
6. xerox.ilu.IluDebugHooks
7. xerox.ilu.IluDebugWriter
8. xerox.ilu.IluExceptionRep
9. xerox.ilu.IluFactory
10. xerox.ilu.IluGCClient
11. xerox.ilu.IluInit
12. xerox.ilu.IluInit2
13. xerox.ilu.IluIOFunctions
14. xerox.ilu.IluMethodArgRep
15. xerox.ilu.IluMethodRep
16. xerox.ilu.IluOInt
17. xerox.ilu.IluServerConnection
18. xerox.ilu.IluSkeleton
19. xerox.ilu.IluWPBase
20. xerox.ilu.IluSurrogateConnection
21. xerox.ilu.IluSurrogateObject
22. xerox.ilu.IluTypeRep
23. xerox.ilu.IluUnixIdentityType

7.9 Full Custom Mapping

Full custom mapping is the ability to externally specify how an isl type shall be mapped to java.

It is similar to custom records and custom objects, however there is no requirement for the custom mapped type to be a subtype of the regular mapping.

There is the restriction that custom mapping works only java object types; not basic types. In particular, the regular mapping of the ilu type needs to map into a java object type, as well as the custom mapped type must be a java object type.

Full custom mapping is used solely in one address space; communicating ilu processes might not know whether or how an ilu type is custom mapped in the other address space.

All stubs in one address space however must agree on the mapping of any ilu type.

Custom mapping is started using isl DOCUMENTATION options or stubber command line options, however there needs to be also runtime code which handles the conversion from and to the regular mapped object.

For full information, check the ILUSRC/etc/javaobv and the ILUSRC/examples/testjavaobv directories.

7.9.0.1 Stubbing

In an ISL interface the custom mapping can be specified with an interface DOCUMENTATION option:

```
DOCUMENTATION "JAVA-CUSTOMFILE" "description-file-name";
```

When stubbing the custom mapping could also be specified with an commandline option

```
-custom description-file-name
```

We are using a file to describe the custom mapping because the actual description too cumberson for command line options and must be set identical for all stubs accessing a custom mapped type.

The description file is a list of lines. Each line is either a comment, or, a custom mapping description for one type. Custom mapping description have five fields separated with white space.

1. 1) The name of the isl interface (which contains the isl type)
2. 2) The name of the isl type to be custom mapped
3. 3) The java class used to map the isl type into
4. 4) The java class which is used for holders
5. 5) If present: The name of a java class which will be loaded by the stub; this is the class which is expected to implement the transformations between wire type and custom type.

7.9.0.2 Runtime features

Tha java interface xerox.ilu.IluCustomMapping describes the functions necessary to implement the custom mapping.

An xerox.ilu.IluCustomMapping object must be registered with the helper class for the regularly mapped type using the static "_registerCustomMapping" method.

Custom mapped type and subclassing do interact whith each other: (Custom mapping works with what ilu calls static types.)

1. -When you receive a subclass of the static ilu-type; that subclass info gets lost
2. -Because of syntax checking you can not transmit a subclass of the static ilu-type, unless it also is a subclass of the java class. However in that case that case subclass'ness gets lost.
3. -when the static class of an argument is a superclass of something with custom mapping on receive: customness is lost and you receive the super class when transmit: syntax checking will not allow this unless custom class is also subclass of static class. (That case is already defined above: subclass'ness gets lost)

7.9.1 Java objects by value, Java marshalled objects

Ilu has a package which supports transmitting arbitrary java objects through ILU. To non-Java applications these look like regular ILU records containing bytes; to Java applications these are very specific Java classes.

This is implemented using full custom mapping using the java type java.lang.Object for ILU types and using java serialization to convert to and from regular ILU records at runtime.

The isl types xerox.ilujava.JavaObject is used to describe an arbitrary java.lang.Object. The java objects assigned to a ILU type ilujava.JavaObject are converted automaticly when transitting or receiving the objects. The Java application never sees the internal representation. Choose objects by value for direct communications between clients and servers which want to deal with life objects.

The isl types xerox.ilujava.MarshalledObject maps into a java object containing the serialized contents of a java object. Java objects are externalized or internalized explicitely before when converting to xerox.ilujava.MarshalledObject. The convertion routines are in the java class xerox.ilujava.IluMarshaller. Choose marshalled objects when the types are use in "dead" services which don't automaticly want to internalize the objects, e.g. For registrations in a name service or database.

Java objects must satisfy one of these conditions to be used as xerox.ilujava.JavaObject or xerox.ilujava.MarshalledObject.

1. Implement the java serializable interface
2. Ilu surrogate objects. Internalizing a MarshalledObject with a surrogate might turn the interned object into an ilu true object.
3. Ilu true objects, if they declare to implement the java serializable interface. The contents of the MarshalledObject will become a surrogate object.
4. Ilu enumerations, arrays, sequences, records, unions, but only if all element types can be serialized. (enumerations do not work correctly in jdk1.1; see javasoft's Bug Id 4163916)
5. The xerox.ilujava.MarshalledObject type itself (rather its custom record aequivalent)

Restriction on Ilu object types: When internalizing, the implementing Java class MUST be loaded. Ilu deserialization of Ilu types can not load implementations: At deserialization time ilu locks may be held which might prevent ilu type registrations. Given this general problem, we don't even bother storing the Java class with the sbh used to represent an ilu object type.

Of course, to a non-java applications both the xerox.ilujava.JavaObject and xerox.ilujava.MarshalledObject types look like their regular ISL records with gibberish sequences of bytes, as internalizing requires a Java VM.

Whether the classes are actually transmitted through the ilu connection or loaded by other means now is part of the state of the ilu-serialized object. The package generalizes over this aspect and class loading schemes can (MUST) be registered. Use the xerox.ilujava.IluMarshaller.setDefaultAccessMethod method.

The xerox.ilujava.IluMarshaller class has methods by which classloader functionality can be registered. (Method: xerox.ilujava.IluMarshaller.registerClassLoaderGetter).

The directory also contains a classloader which transmits class bytes through the same ilu connection. This classloader is unsafe, unless you can guarantee the true object of the classloader only sends valid and safe class bytes. Arbitrary mechanisms and security can be implemented using the registration mechanisms.

Another class loader is provided which loads the code through an URL. (This loader requires JDK1.2)

A more ideal world would decide on objects by value versus marshalled objects semantics on a more dynamic base. This is however difficult as they would need to be stubbed differently. Current ILU supports some choice dymamicly as it allows to transmit a xerox.ilujava.MarshalledObject where a xerox.ilujava.JavaObject is expected. In this case the other end will get a xerox.ilujava.MarshalledObject as expected; that is the reason xerox.ilujava.MarshalledObject are `"serializable"`.

Users of Java objects by value or Java marshalled objects must make the classes available, e.g by putting the ILUHOME/lib/ilujavaobv.jar jar file onto the classpath.

For more information, check the ILUSRC/etc/javaobv directory.

An example application is in the ILUSRC/examples/testjavaobv directory. The demo program shows how to transmit arbitrary (though serializable) java objects.

This application even transmits Java classes (byte codes) through ilu. Unlike RMI, the class itself is passed through ILU. We do not claim the simpler mechanism of RMI wouldn't be sufficient, but by specifying access of the byte codes we can better show the power of the mechanisms.

7.9.1.1 Interoperability of Java objects by value, Java marshalled objects

ILU-serialization and Java-serialization

Do NOT confuse ILU's serialization to with java's standard serialization. The two mechanism are different, however they DO interact.

Ilu stubs do implement java.io.Serializable. This is in support of this package. Java-serialization and Java-de-serialization for some ILU types depends on substitution performed in this package and may not work with generic java-serializable applications, ie. RMI. Most ILU generated stubs work perfectly with RMI.

Java-Deserialization requires ILU being loaded.

Most important compatibility: Any java-serializable object can be ilu-serialized. The ILU MarshalledObject type is java-serializable.

ILU-serialization and Corba compatibility

The serialization adds features to ILU stubs. They do not interfere with the source code level compatibility. However standard Corba stubs will not have these features and would interfer with ILU-serialization.

The ISL types supporting serialized ILU objects contain no feature which couldn't be represented with standard IDL files, but the IDL file has not yet been written.

ILU-serialization and non-java ILU applications

To other other programming languages ILU-serialized objects look like ILU records with fields and sequences of bytes containing gibberish. The important point is: Such records can be handled, stored, and, passed along and become life again when forwarded into ILU for Java.

8 Using ILU with Common Lisp

8.1 Introduction

This document is for the Common Lisp programmer who wishes to use ILU. The following sections will show how ILU is mapped into Common Lisp constructs and how both Common Lisp clients and servers are generated and built.

8.2 The ISL Mapping to Common Lisp

Runtime code is in the Common Lisp package `ilu`.

8.2.1 Names

Names from interface specifications are transformed into Lisp names (case-insensitive) by inserting hyphens at lower-to-upper case transitions. Hyphens that are already present are maintained as is.[1] Method names are mapped to `INTERFACE:TYPE.METHOD`.

8.2.2 Interface

A separate package is defined for each interface with `defpackage`. The name of this package is taken from the name of the interface. This package uses the packages `common-lisp` and `ilu`. The Common Lisp names of all entities defined in the ISL are exported from the package, including types, classes, constants, accessors, type predicates, generic functions, exceptions, etc. Such symbols are also shadowed, to avoid conflicts with used packages. For example, given the following interface:

```
INTERFACE MyInterface END;
EXCEPTION TotalWinner : Person;
TYPE Person = OBJECT
  METHODS
     Enemies (someone : Person) : Cardinal
       RAISES TotalWinner END
  END;
```

the stubber generates the following `defpackage`:

```
(defpackage :my-interface
  (:use :common-lisp :ilu)
  (:shadow #:total-winner #:person #:person.enemies)
  (:export #:total-winner #:person #:person.enemies)
```

---

[1] This causes problems; the ISL names "FooBar" and "foo-bar" map to the same Common Lisp name. Something will have to change.

This allows symbols defined in the `commonlisp` package to be used by the automatically generated code in the generated package, but it also means that the user needs to be careful about using any generated package. In general, we recommend that you explicitly specify the full name of symbols from ILU interfaces.

8.2.3 Basic Types

The basic ISL types have the following mapping to Common Lisp types:

- ISL `BOOLEAN` maps to Common Lisp `(or nil t)`
- ISL `BYTE` maps to Common Lisp `(unsigned-byte 8)`
- ISL `SHORT CARDINAL` maps to Common Lisp `(unsigned-byte 16)`
- ISL `CARDINAL` maps to Common Lisp `(unsigned-byte 32)`
- ISL `LONG CARDINAL` maps to Common Lisp `(unsigned-byte 64)`
- ISL `SHORT INTEGER` maps to Common Lisp `(signed-byte 16)`
- ISL `INTEGER` maps to Common Lisp `(signed-byte 32)`
- ISL `LONG INTEGER` maps to Common Lisp `(signed-byte 64)`
- ISL `CHARACTER` maps to Common Lisp `character`
- ISL `SHORT CHARACTER` maps to Common Lisp `character`
- ISL `SHORT REAL` maps to Common Lisp `single-float`
- ISL `REAL` maps to Common Lisp `double-float`
- ISL `LONG REAL` maps to Common Lisp `double-float`

8.2.3.1 Constant

Constants are implemented in CL by a value of the appropriate type, defined with `defconstant`.

8.2.4 Strings

Arrays and sequences of CHARACTER (regular or SHORT) are implemented as `simple-strings`.

8.2.5 Pickles and Typecodes

Pickles are represented with the CLOS class `ilu:pickle`

Instances of a pickle may be created by calling `cl:make-instance` on `ILU:PICKLE`, with the `:VALUE` and `:TYPE` keywords, as in
`(cl:make-instance 'ilu:pickle :type 'ilu:cardinal :value 3456)`

Pickle has three reader functions defined on it:

- `ilu:pickle-value` returns the value stored in the pickle as a Lisp value.
- `ilu:pickle-type` returns the type of the value stored in the pickle.
- `ilu:pickle-bytes` returns the pickled bytes of the pickle.

8.2.6 Constructed Types

8.2.6.1 Enumeration

Enumerations are implemented with symbols, as in
`(deftype answer () '(member 'yes 'no 'maybe))`

8.2.6.2 Array

Arrays are implemented as `simple-arrays`.

8.2.6.3 Sequence

Sequences are implemented as `lists`, except for sequences of characters, which are implemented as `simple-strings`.

8.2.6.4 Record

Record types are implemented with CL `defstruct`.

8.2.6.5 Union

Unions are implemented as a cons'ed value, with the cdr containing the union type discriminant, and the cdr containing the actual value.

8.2.6.6 Optional

8.2.7 Object Types

Classes are implemented with CLOS `defclass`.

Private slots are created for methods which are specified as `functional`, and the runtime caches the value of this method in such slots after the first call to the method.

Instances are always subtypes of `ilu:ilu-object`.

Methods always take as their first argument the object which they are a method on. Subsequent arguments are those specified in the '`.isl`' file. Methods that have `OUT` or `INOUT` arguments may return multiple values. In general, the parameters to a method are the `IN` and `INOUT` parameters specified in the ISL interface, but not the `OUT` parameters. The return values from a method are the specified return value for the ISL method, if any, followed by the `INOUT` and `OUT` parameters for the method, if any, in the order in which they appear in the ISL specification of the method.

OMG IDL attributes map to a CLOS method of the same name, and a setf method with the same name (unless the attribute is readonly).

8.2.7.1 Surrogate and True Object Types

8.2.7.2 Methods, Parameters, and Exceptions

Exceptions are represented with CL conditions, defined by `define-condition`. All ILU conditions are subtypes of `ilu:rpc-exception`, which is a `serious-condition`. If an associated value is specified for an exception it may be accessed in one of the following two ways:

1. If the name of the value type begins with `"-ilu-prefix-idlExceptionType-"`, the value type is a generated type from an OMG IDL exception description, and is a record type. In this case, each of the fields of the record type are placed in the condition individually, and an accessor with that field name is declared for that field.
2. In all other cases, there is a single accessor called `value` through which the associated value may be read.

8.2.7.3 Garbage Collection and COLLECTIBLE

8.3 Access to standard ILU features

8.3.1 Servers and Ports

8.3.1.1 Object Tables

8.3.1.2 Server Relocation

8.3.2 Threading and Event Loops

8.3.3 Custom Records

8.3.4 Custom Surrogates

Custom surrogates allow the user to specify custom surrogate object types which may have additional functionality in terms of caching or other side effects, and have them created instead of the default ILU surrogate object type when an instance is received. This functionality is provided in the Common Lisp runtime with the function `ilu:register-custom-surrogate`.

8.3.5 String Binding Handle Formation

8.3.6 Simple Binding

8.3.7 Principal Identities and Passports

It's also possible to find out who is making the call by examining the value of `ilu:*caller-identity*`.

8.4 Building Common Lisp/ILU Applications

8.4.1 Stub Generation

The program ILU `lisp-stubber` takes a interface specification (an '`.isl`' file) and generates lisp code to provide both client-side and server-side support for the interface. The files are generated in the current working directory. In particular, the following files are generated:

- '*interface-name*`-sysdcl.lisp`' -- tells `PDEFSYS`[2] how to compile and load the other files. It defines a Common Lisp module `:`<*interface*>, which describes the code needed to support both surrogate and true use of the interface. This file is often called a *sysdcl* for the module.
- '*interface-name*`-basics.lisp`' -- contains lisp code needed by clients of the module; and
- '*interface-name*`-server-procs.lisp`' -- contains lisp code needed by module implementations.

8.4.2 Implementing an ILU module in Common Lisp

For each ILU class *interface*`.`*otype*, ILU will define, in the file '*interface*`-server-procs.lisp`', a CLOS class called *interface*`:`*otype*`.IMPL`. To implement a true object for *interface*`.`*otype*, one should further subclass this CLOS class, and override all of its methods. In particular, do not let any of the default methods for the class be called from your methods for it.

ILU supports, in each address space, multiple instances of something called a *kernel server*, each of which in turn supports some set of object instances. A kernel server *export*s its objects by making them available to other modules. It may do so via one or more *port*s, which are abstractly a tuple of (*rpc protocol*, *transport type*, *transport address*). For example, a typical port might provide access to a kernel server's objects via `(Sun RPC, TCP/IP, UNIX port 2076)`. Another port on the same kernel server might provide access to the objects via `(Xerox Courier, XNS SPP, XNS port 1394)`.

When creating an instance of a true object, a kernel server for it, and an instance id (the name by which the kernel server knows it) for it must be determined. These may be specified explicitly by use of the keyword arguments to `commonlisp:make-instance :ilu-kernel-server` and `:ilu-instance-handle`, respectively. If they are not specified explicitly, the variable `ilu:*default-server*` will be bound, and its value will be used; a default instance handle, unique relative to the kernel server, will be generated.

---

[2] See Section C.1 [The ILU Common Lisp Portable DEFSYSTEM Module], page 300, for a description of the `PDEFSYS` package.

A kernel server may be created by instantiating the class `ilu:kernel-server`. The keyword argument `:id` may be specified to select a name for the server. Note that ILU object IDs, which consist of the kernel server ID, plus the instance handle of the object on that server, must be unique ‘‘across space and time’’, as the saying goes. If no kernel server id is specified, ILU will generate one automatically, using an algorithm that provides a high probability of uniqueness. If you explicitly specify a kernel server ID, a good technique is to use a prefix or suffix which uniquely identifies some domain in which you can assure the uniqueness of the remaining part of the ID. For example, when using ILU at some project called NIFTY at some internet site in the IP domain `department.company.com`, one might use kernel server IDs with names like `something.NIFTY.department.company.com`.

```
=> (make-instance 'ilu:kernel-server :id "FOO-SERVER-1")
#<ILU:KERNEL-SERVER "FOO-SERVER-1">
=> (make-instance 'ilu:kernel-server)
#<ILU:KERNEL-SERVER "121.2.100.231.1404.2c7577eb.3e5a28f">
=>
```

8.4.2.1 Implementation Inheritance

8.4.2.2 Exporting Objects

To export a module for use by other modules, simply instantiate one or more instances of your subtypes of *interface*`:`*otype*`.IMPL` (which will inherit from `ilu:ilu-true-object`.

```
=> (make-instance 'foo:my-bar.impl :ilu-kernel-server s)
#<FOO:MY-BAR.IMPL 0x3b32e8 "1">
=>
```

The simplest Common Lisp ‘‘server’’ code would look something like:

```
(defun start-server ()
  (make-instance 'foo:my-bar.impl))
```

which will create an instance of `FOO:MY-BAR.IMPL` and export it via a default server.

To enable users of your module find the exported objects, you may register the string binding handle of the object or objects, along with their type IDs, in any name service or registry that is convenient for you. In release 1.6 of ILU, we are supporting an experimental simple binding method that allows you to ‘‘publish’’ an object, which registers it in a domain-wide registry, and then to withdraw the object, if necessary. Potential clients can find the string binding handle and type ID of the object by calling a lookup function. **Note that this interface and service is experimental, and may be supported differently in future releases of the ILU system.**

If you wanted to create an instance, and publish it, the code for starting a service might look something like this:

```
(defun start-server ()
```

```
(let* ((ks (make-instance 'ilu:kernel-server
                ;; specify the service id
                :id "service.localdomain.company.com"))
       (si (make-instance 'foo:my-bar.impl
                ;; specify the server
                :ilu-kernel-server ks
                ;; specify the instance handle
                :ilu-instance-handle "theServer")))
  ;; the OID for "si" is now "theServer@service.localdomain.company.com"
  (ilu:publish si)
  si))
```

Someone who wanted to use this service could then find it with the following:

```
(defun find-server ()
  (ilu:lookup 'foo:bar "theServer@service.localdomain.company.com"))
```

8.4.2.3 Debugging

To help with finding errors in your methods, the variable `*debug-uncaught-conditions*` is provided.

8.4.3 Using an ILU module in Common Lisp

To use a module from Common Lisp, you must first have loaded the PDEFSYS ‘‘system’’ that describes the module. Typically, for an ILU interface called *Foo*, the system can be loaded by invoking `(pdefsys:load-system :foo)`. Next, you must bind an instance of an object from that interface. The most common way of doing this is to receive an instance of an object from a method called on another object. But to get the first object exported by that module, one can use either `ilu:sbh->instance` or `ilu:lookup`.

8.4.4 Dumping an image with ILU

ILU has dynamic runtime state. In particular, after it is initialized, it uses several Common Lisp threads to maintain part of its state, and may also keep open connections on operating system communication interfaces. If you wish to dump an image containing ILU, you must dump the image before initializing the `ILU` module.

Initialization occurs automatically whenever a instance of `ilu:ilu-object` or `ilu:rpc-server` is created. Thus you should not create any instances of either true or surrogate ILU objects before dumping the image. However, you may load all the interface code for any interfaces that you are using, before dumping the image.

Initialization may also be accomplished by an explicit call to `ilu:initialize-ilu`. You may check to see whether the system has been initialized by examining the variable `ilu::*ilu-initialized*`, which is `t` iff `ilu:initialize-ilu` has been invoked.

8.4.5 Notes for Microsoft Windows Users

8.4.5.1 Installation on the MS Windows platform

To install the Lisp binding on the MS Windows platform proceed as follows: Copy or rename the file '`ilu-non-threaded-sysdcl.lisp`' in directory '*ILUSRC*`\runtime\lisp`' to '`ilu-sysdcl.lisp`'. Compile the Lisp runtime files; i.e., start Allegro and type:

```
(load "c:\\ilu\\src\\runtime\\lisp\\compile-files.lisp")
```

Copy the resulting '`*.fsl`' files and the files '`ilu-sysdcl.lisp`' and '`pdefsys.lisp`' to the Lisp installation directory ('*ILUHOME*`\lisp`'). Copy the ILU kernel and Lisp DLLs into a directory that is on your `PATH`.

8.4.5.2 Allegro 3.0.1 Needs `ilu:run-main-loop`

Because Allegro 3.0.1 is single-threaded, servers on Windows 95/NT must run the ILU mainloop. To run it indefinitely, use

```
(ilu:run-main-loop)
```

Or allocate a handle, which can later, presumably in a method call, be used to exit the event loop:

```
(setf *handle* (ilu:create-main-loop-handle))
(ilu:run-main-loop *handle*)
...
(ilu:exit-main-loop *handle*)
```

For example, to run the example server in directory '`examples/test1`', start Allegro 3.0.1 for Windows, and type the following:

```
(load "c:\\ilu\\examples\\test1\\load-lisp-example.lisp")
(test1-server:start-server)
(ilu:run-main-loop)
```

8.4.6 The Portable DEFSYSTEM Module

ILU support uses a portable implementation of `DEFSYSTEM` to specify modules to Common Lisp. See Section C.1 [The ILU Common Lisp Portable DEFSYSTEM Module], page 300, for details of this system.

8.4.7 ILU Common Lisp Lightweight Processes

ILU currently assumes the existence of lightweight process, or thread, support in your Common Lisp implementation. It uses these internally via a generic veneer, described fully in Section D.11 [The ILU Common Lisp Lightweight Process System], page 319.

8.4.8 Porting ILU to a New Common Lisp Implementation

The Lisp support provided with ILU includes support for the Franz Allegro Common Lisp 4.x implementation. To use ILU with other Common Lisp implementations, please see Section E.6 [Porting ILU to Common Lisp Implementations], page 325.

8.5 Common Lisp/ILU API Reference

ilu:ilu-class-info (*DISC* (`or ilu:ilu-object type-name`)) (*WHAT* `keyword`) Method
=> (`or string boolean list`)

This routine will return the specified piece of information about the ILU class specified with *DISC*, which may be either a CLOS class name, or an instance of the class, and with *WHAT*, which identifies which piece of information to return. *WHAT* may have the following values:

- `:authentication` -- what kind of authentication, if any, is expected by the methods of this class
- `:brand` -- the brand of the object type, if any
- `:collectible-p` -- whether or not the object type participates in the ILU distributed GC
- `:doc-string` -- the doc string specified for the object type
- `:id` -- the ILU unique ID for the object type
- `:ilu-version` -- which version of ILU the stubber that generated the code for this object type came from
- `:methods` -- a list of the methods of the object type
- `:optional-p` -- whether values of this class are allowed to be `cl:nil` (a CORBA excrescence)
- `:name` -- the ILU name of the object type

cl:make-instance 'ilu:kernel-server &key (*id* `string nil`) (*unix-port* `fixnum 0`) (*object-table* `list of 2 elements nil`) (*protocol* `string "sunrpc"`) (*transport* `list of string ("sunrpcrm" "tcp_0_0")`) => `ilu:kernel-server` Method

Creates and returns an instance of `ilu:kernel-server`. If *id* is specified, the server has that value for its server ID. If *unix-port* is specified, the server attempts to 'listen' on that UNIX port, if the notion of a UNIX port is applicable. If *object-table* is specified, it must consist of a list of two functions. The first function must take a string, which is the instance handle of a desired object on this kernel server, and return a value of type `ilu:ilu-true-object`. The second funtion must free up any resources used by this object table. Specific protocols and transport stacks can be specified with the `protocol` and `transport` keywords; these default to whatever defaults were selected when your ILU installation was built.

cl:make-instance 'ilu:ilu-true-object &key (*ilu-kernel-server* `ilu:kernel-server nil`) (*ilu-instance-handle* `string nil`) => `ilu:ilu-true-object` — Method

Creates and returns an instance of `ilu:ilu-true-object`. If *ilu-true-server* is specified, the instance is created on the specified server. If *ilu-instance-handle* is specified, that instance handle is used.

ilu:*caller-identity* — Variable

The identity of the caller is bound to the special variable `ilu:*caller-identity*`. It is a string which begins with the name of an identity scheme, followed by an identity in that scheme. For example, an identity in the SunRPC UNIX identity scheme would be something like `"sunrpc-unix:2345,67@13.12.11.10"` (i.e., `"sunrpc-unix:`<*uid*>`,`<*gid*>`@`<*hostname*>`"`). If no identity is furnished, a zero-length string is bound.

ilu:publish (*OBJ* `ilu:ilu-object`) => `boolean` — Function

Accepts an `ilu:ilu-object` instance and registers it with some domain-wide registration service. The object is known by its *object ID* (OID), which is composed of the ID of its kernel server, plus a server-relative instance ID, typically composed as *instance-ID*`@`*server-ID*. Clients may find the object by looking up the OID via the `ilu:lookup` function. The function returns non-`cl:nil` if the publication succeeded.

ilu:withdraw (*OBJ* `ilu:ilu-object`) => `boolean` — Function

If *OBJ* is registered, and if it was published by the same address space that is calling `withdraw`, its registration is withdrawn. The function returns non-`cl:nil` if the object is no longer published.

ilu:*debug-uncaught-conditions* — Variable

If `cl:t`, causes a server to invoke the debugger when an unhandled error in user code is encountered, rather than the default action of signalling an exception back to the caller. The default value is `cl:nil`.

ilu:register-custom-surrogate (*CLASS-NAME* `symbol`) (*CUSTOM-CLASS* `clos:standard-class`) — Function

Instructs the runtime to create an instance of *CUSTOM-CLASS* whenever it would normally create a new instance of the ILU object type named by *CLASS-NAME*, which should be the Common Lisp name for the object type. *CUSTOM-CLASS* must be a subtype of the class named by *CLASS-NAME*.

ilu:initialize-ilu — Function

Initializes the `ILU` module.

ilu:sbh->instance (*PUTATIVE-TYPE-NAME* `symbol`) (*SBH* `string`) Function
&optional (*MOST-SPECIFIC-TYPE-ID* `simple-string mstid of specified PUTATIVE-TYPE`) => `ilu:ilu-object`

Accepts an ILU string binding handle and Common Lisp type name, and attempts to locally bind an instance of that type with the OID specified in the string binding handle. If no such instance exists locally, a surrogate instance is created and returned. If a true instance exists locally, that instance will be returned.

ilu:lookup (*server-id* `simple-string`) (*instance-handle* `simple-string`) Function
(*PUTATIVE-TYPE-NAME* `symbol`) => (`or nil ilu:ilu-object`)

This routine will find and return an object with a server ID of *server-id* and instance handle of *instance-handle*, if such an object has been registered in the local domain via the ILU simple binding protocol. See the section on ''Exporting Objects'' for an example.

ilu:ping (*DISC* `ilu:ilu-object`) => (`or t nil`) Method

Returns `cl:t` if the true object for *DISC* exists, and the process serving it can be contacted; `cl:nil` otherwise.

9 Using ILU with Scheme (Guile)

9.1 Introduction

This document is for the Scheme programmer who wishes to use ILU. The following sections will show how ILU is mapped into Scheme constructs and how both Scheme clients and servers are generated and built.

The Scheme support in ILU was kindly donated by
Siemens Corporate Research Inc. (`http://www.scr.siemens.com/`), andwritten by Bill Nell (`mailto:bnell@scr.siemens.com`).

9.2 The ISL Mapping to Scheme

9.2.1 Names

The mapping of ISL names to Scheme names for objects and methods are described in the following sections. For all other ISL types and exceptions the Scheme name will be *module-name*:name.

9.2.2 Basic Types

This section describes the mapping of ISL types to Scheme types. Note that some mappings use extra precision where it is not necessary. This is because the Scheme types have no finer distinctions for these types. The basic ISL types have the following mapping to Scheme constructs:

- ISL `boolean` is mapped to Scheme `boolean`;
- ISL `short cardinal` is mapped to Scheme `unsigned long`;
- ISL `cardinal` is mapped to Scheme `unsigned long`;
- ISL `long cardinal` is mapped to Scheme `pair (unsigned long . unsigned long)`;
- ISL `short integer` is mapped to Scheme `long`;
- ISL `integer` is mapped to Scheme `long`;
- ISL `long int` is mapped to Scheme `pair (long . unsigned long)`;
- ISL `short real` is mapped to Scheme `double precision real`;
- ISL `real` is mapped to Scheme `double precision real`;
- ISL `short character` is mapped to Scheme `character`;
- ISL `character` is mapped to Scheme `long`;
- ISL `byte` is mapped to Scheme `character`.

9.2.2.1 Constant

9.2.3 Strings

- ISL `string` is mapped to Scheme `string`.
- ISL `wstring` is mapped to Scheme `vector of unsigned longs`.

9.2.4 Pickles and Typecodes

Not implemented yet.

9.2.5 Constructed Types

9.2.5.1 Enumeration

Implemented with a set of symbols bound to the integer values of the enumeration type.

9.2.5.2 Array

Arrays are implemented as Scheme `vectors`. For arrays, a Scheme function will be generated that takes no arguments and returns a vector with dimensions corresponding to the ISL array type. The user is responsible for setting elements in the vector.

9.2.5.3 Sequence

Sequences are implemented as `lists`, except for sequences of characters, which are implemented as `strings`.

9.2.5.4 Record

Each record is mapped to an ILU-YASOS object. Each object has a constructor named (make-*module-name*:*record-name*) that takes no arguments. Each object also has methods to get and set all the fields of that record. These accessors follow the pattern of (get-*field-name* obj) and (set-*field-name* obj value). In the future record constructors will be able to take arguments to initialize their fields.

9.2.5.5 Union

Unions are implemented as a cons'ed value, with the cdr containing the union type discriminant, and the cdr containing the actual value. For union types, a constructor function taking two arguments is created. The first argument is the discriminator and the second is the union value. The discriminator of a union can be accessed or set using the `car` and `set-car!` functions, respectively. The value of a union can be accessed or set using the `cdr` and `set-cdr!` functions, respectively.

9.2.5.6 Optional

Either `#f` or the value. Note that this mapping is broken for optional boolean types.

9.2.6 Object Types

Each object is mapped to an ILU-YASOS object.

Each object type has a constructor named (make-*module-name*:*object-name*) that takes no arguments. The user should not attempt to create objects using the constructor since these are surrogate objects and not true objects. All the method names should map exactly as they appear in the ISL definition.

9.2.6.1 Surrogate and True Object Types

ilu:object Class

This is the root class of all objects in the Scheme LSR. It provides some basic functionality required by all ILU objects. The public methods are described below. Since ILU-YASOS has no notion of public or private there are additional methods which are not described here and should not ever be called by the user.

- `(string-binding-handle` *obj*`)` - return the string binding handle of *obj*
- `(publish` *obj*`)` - publish this object. Returns a boolean value for success or failure.
- `(withdraw` *obj*`)` - withdraw this object. Returns a boolean value for success or failure.
- `(class-name` *obj*`)` - return the class name for this object.
- `(class-id` *obj*`)` - return the class id for this object.
- `(get-server` *obj*`)` - get the kernel server that controls this object.
- `(get-instance-handle` *obj*`)` - get the instance handle for this object.
- `(get-kernel-server` *obj*`)` - get the true server for this object.
- `(get-instance-class-record` *obj*`)` - get the instance class record for this object.
- `(destroy` *obj*`)` - destroy this surrogate object. Unfortunately, ILU-YASOS objects are not hooked into the scheme garbage collector yet. So, the destroy method must be called when you are finished with a surrogate object to prevent memory leaks.

Surrogate objects present an interface to access a true objects which may or may not live in the same address space as the surrogate object. The user is not allowed to create their own surrogate objects. They must be looked up using a name binding service or through the use if a string binding handle.

Behind every surrogate object there must be a true object. The user is in charge of defining true objects. Implementing a true object for a particular surrogate object is accomplished by subclassing the surrogate object class provided by the Scheme stubber. Each true object implementation must override every method of the surrogate `"parent"` object to work correctly. See the section on ILU-YASOS for implementing objects in Scheme. Also, see the examples of a Scheme client and server given in the examples/test1 directory.

IMPORTANT CAVEAT: When using multiple inheritance with ILU-YASOS objects to implement `"true"` objects it is important that the correct ILU class record be assigned to the true object. The class record of the last class listed in the superclass list of the object definition will be the one used for the object being defined. If this is not the desired class record, the implementor must set it by hand. (Hopefully, in the future this will be taken care of automatically). Usually, this is not a problem if you always place the most specific superclass of an object last in the list of superclasses.

9.2.6.2 Methods, Parameters, and Exceptions

All methods take at least two arguments in addition to any other arguments specified in the ISL definition. The first argument is the object the method is being called on and the second argument is the status object. The remaining arguments are the same as in the ISL definition.

ilu:status Class

This is a status object type, used to record the success or failure of all method calls. Later this will be replaced with catch and throw. Also note that even though individual modules create their own status types, the ilu:status type can still be used in their place.

`ilu:status` objects support the following methods:

- `(get-return-code` *this*`)` - get the return code for this status object. If the value is ilu:success then the method call was successful. Any other value indicates an exception string, which can be printed.
- `(set-return-code` *this* `code)` - set the return code of this status object.
- `(get-status-value` *this*`)` - get the value associated with any exception raised in this status object. If the call was successful this field should be #f. Otherwise it will contain the contents of an exception record.
- `(set-status-value` *this* `value)` - set the status value of this status object. Used for setting additional information when an exception is raised.
- `(get-caller-passport` *this*`)` - get the passport for this method call.
- `(set-caller-passport` *this* `passport)` - set the passport for this method call.

Normal methods are called in the following manner:
`(`*method-name object status arguments*`)`
If any errors are encountered while executing the method, they will be stored in the status object parameter. Failed method calls always return `#f`.

Arguments defined as "out" must still be passed as placeholders even though their values are ignored. The return values of methods are as follows: If the method has no "inout" or "out" parameters, a single value is returned. When a method has "inout" or "out" parameters the method returns a list of values which contains any "inout" and "out" values (in the order they are specified in the ISL definition) followed by the "normal" return value as the last element of the list.

All exceptions are raised by setting the return code of a status value to something other than ilu:success. Depending on the type of the exception there may also be additional data associated with it that can be set using the "set-status-value" method on a status object.

Asynchronous methods are available in the Scheme LSR. They will return immediatedly with a return value of #t.

Functional methods are currently not supported in the Scheme LSR. At the moment they are treated as normal method calls.

9.2.6.3 Garbage Collection and COLLECTIBLE

At the moment garbage collection of surrogate and true objects is not supported.

9.3 Access to standard ILU features

9.3.1 Servers and Ports

9.3.1.1 Object Tables

See `ilu:create-object-table`.

9.3.2 Threading and Event Loops

At the moment the Scheme LSR does not support threading, so only the event loop mode of operation will work.

9.3.3 Custom Records

9.3.4 String Binding Handle Formation

9.3.5 Simple Binding

9.3.6 Principal Identities and Passports

9.4 Building Scheme/ILU Applications

9.4.1 Initialization order

All object and type definitions are initialized automatically when loading the Scheme code for a particular interface. The (ilu:init) function is used to initialize the GC server and callback object. If the user wishes to select their own main loop object, they must register it before calling (ilu:init).

9.4.2 ILU-YASOS

The ILU version of YASOS is a slightly modified version of the standard YASOS with the following differences:

- All the standard YASOS functions are prepended with the prefix `"ilu-"`, so regular YASOS and ILU-YASOS can operate together.
- All ILU-YASOS methods make take a variable number of arguments. This is so the same method may take different numbers of arguments depending on which object it is called with.
- An additional function called ilu-bind-arguments is provided to extract multiple arguments from a list and bind them to symbols (almost like a let block). It is provided only as a convienience and is not a required part of a method definition.

9.4.3 Where to get Guile 1.2 and SLIB

Guile 1.2 (`ftp://prep.ai.mit.edu/pub/gnu/guile-1.2.tar.gz`).
SLIB 2b1 (`http://www-swiss.ai.mit.edu/~jaffer/SLIB.html`).
Guile snapshots (`ftp://ftp.red-bean.com/pub/guile/guile-snap.tar.gz`) -- Guile snapshots are daily updates of the Guile system, use them at your own risk. Also some snapshots may not be compatible with the current implementation of the stubber and runtime.

See also the Report on the Algorithmic Language Scheme -- revision 4 (`http://www-swiss.ai.mit.edu/~jaffer/r4rs`
and the SLIB Reference Manual (`http://www-swiss.ai.mit.edu/~jaffer/slib_toc.html`).

9.4.4 Stub Generation

9.4.5 Implementing an ILU module in Scheme

9.4.5.1 Implementation Inheritance

9.4.5.2 Exporting Objects

9.4.6 Using an ILU module in Scheme

If you are using Guile Scheme, the value of the environment variable `LD_LIBRARY_PATH` should include the directory in which the `ilu` library for Guile has been installed; that's normally '*ILUHOME*`/lib`'. Additionally, the environment variable `SCHEME_LOAD_PATH` should contain '*ILUHOME*`/guile`', so that the ILU Guile files can be found.

9.5 Scheme/ILU API Reference

- `(ilu:init)` - called to initialize the scheme interface to ILU. If the user wants to set their own main loop (see (ilu:set-main-loop)), they must do so before calling (ilu:init). Multiple calls to (ilu:init) will have no effect.
- `(ilu:time-now)` - get the current time in seconds as a floating point value.
- `(ilu:find-class-from-type-name name)` - get the class record given the type name of an object.
- `(ilu:find-class-from-id id)` - get the class record given the type id of an object.
- `(ilu-class:name class)` - given a class record object, get the name of the class.
- `(ilu:create-object-table object-of-ih-func free-self-func)` - create an object table given a function that converts instance handles to objects and a function that cleans up an object table. object-of-ih-func must take at least two arguments the first being the object table and the second the instance handle of the object being requested. The free-self-func must take at least one argument which is the object table being freed.
- `(ilu:set-default-server server)` - set the default server in the process to be "server".
- `(ilu:get-default-server)` - return the default server object.
- `(ilu:create-port server protocol-type transport-type)` - create a port on the given server. protocol-type is a string containing the protocol type and transport-type is a vector containing the transport layers to use on the given port. protocol-type and transport-type may have the value of #f in which case the default protocol and transport types are used.
- `(ilu:set-server-default-port server port)` - set the default port for a given server.
- `(ilu-server:create server-name object-table)` - create a server given a name and an object-table.
- `(ilu-server:id server)` - return the id of a server.
- `(ilu-server:add-port server protocol-type transport-type make-default)` - add a port to the given server. Basically, the same as "ilu:create-port", except it takes one additional argument "make-default" which indicates whether the new port should be made the default one for the given server.
- `(ilu:make-main-loop-id)` - make an "id" that can be used to start and stop a main loop execution.
- `(ilu:free-main-loop-id id)` - free a main-loop-id created by the function (ilu:make-main-loop-id). This should only be used when the id is no longer needed. Note: the same main loop id can be reused for different main loops if desired, but may not be used by multiple main loops at the same time.
- `(ilu:run-main-loop id)` - run the ILU main loop. id must be 0 or an id created by (ilu:make-main-loop-id). If the id is 0 the scheme runtime will assign a default id to use. Note: that the 0 id can be only used for one main loop at a time.
- `(ilu:exit-main-loop id)` - stop execution of a main loop with the given id.
- `(ilu:set-main-loop main-loop)` - set the main loop to a user defined main loop object. See (make-ilu:main-loop) for more information.
- `(ilu:register-input-handler port handler)` - register in input handler for the given input port. handler may be any lambda expression.

- `(ilu:unregister-input-handler port)` - unregister an input handler for the given port.
- `(ilu:register-output-handler port handler)` - register in output handler for the given output port. handler may be any lambda expression.
- `(ilu:unregister-output-handler port)` - unregister an output handler for the given port.
- `(ilu:create-alarm)` - create an alarm object.
- `(ilu:set-alarm alarm time proc)` - set an alarm object. The time is given as a floating point value in seconds. proc is an arbitrary lambda expression.
- `(ilu:clear-alarm alarm)` - clear the given alarm so it will not be called.
- `(make-ilu:main-loop)` - this is an ILU-YASOS object is an abstract base class for main loop objects. If the user desires to make their own main loop object, the must subclass from this one and redefine all of the following methods.

  Public Methods:

  - `(run this stop)` - run this main loop. stop is a main-loop-id as described in (ilu:make-main-loop-id);
  - `(exit this stop)` - stop this main loop. stop is a main-loop-id as described in (ilu:make-main-loop-id);
  - `(register-input-handler this port handler)` - register an input handler for the given port. handler can be an arbitrary lambda expression;
  - `(register-output-handler this port handler)` - register an output handler for the given port. handler can be an arbitrary lambda expression;
  - `(unregister-input-handler this port)` - unregister an input handler for the given port;
  - `(unregister-output-handler this port)` - unregister an input handler for the given port;
  - `(create-alarm this)` - create an alarm object;
  - `(set-alarm this alarm time proc)` - set an alarm object (see ilu:set-alarm for more details);
  - `(clear-alarm this alarm)` - clear the given alarm so it won't trigger.
- `(ilu:sbh-to-object sbh class)` - takes a string binding handle and a class record and returns the object associated with that string binding handle. It also checks to make sure the correct class is set.
- `(ilu:parse-sbh sbh)` - parse the given string binding handle. A list is returned which has the following form: (instance-handle server-id mstid contact-info length-of-contact-info-substring).
- `(ilu-object:lookup server-id instance-handle class)` - attempt to lookup an object given a server-id, instance-handle and class record. If the lookup fails, #f is returned.

10 Using ILU with Microsoft Windows

Note: In this document, when you see a reference to Windows, it applies to Windows 95, Windows NT 3.5 and Windows NT 4.0. Windows 3.1 is no longer supported, since ILU makes assumptions about the size of items that can be copied via certain system/library calls (segment sizes are an issue). For this reason, ILU under Windows 3.1 will only work reliably in all situations with other Windows 3.1 systems running ILU.

10.1 Prerequisites for using ILU with Microsoft Windows

10.1.1 Using ILU applications on Windows NT and Windows 95

Windows must be set up to use `TCP/IP`. Use the `Network Configuration and Control` Applet under the Windows NT control panel to install and configure your `TCP/IP` setup. For Windows 95, use the `Network` applet. (See your Windows documentation for further details.) Try all the usual `TCP/IP` applications (e.g. `ping`, `ftp`, `telnet`) to ensure your `TCP/IP` is working properly.

You will also need the redistributable Microsoft C Runtime dynamic link library for NT (for example, '`MSVCRT20.DLL`' if using Visual C++ 2.0 or '`MSVCRT40.DLL`' if using Visual C++ 4.0) on the system, etc.. The Visual C++ redistributable files are located in a '`REDIST`' directory on the Visual C++ CD-ROM disc. Note that versions of Visual C++ later than 4.0 actually have additional DLLs ('`MSVCRT.DLL`') that contain the "real" runtime library. If you've installed Visual C++, then most likely you have the necessary DLLS installed. [Note - you can determine what dlls a dll or exe imports by using the dumpbin utility that comes with Visual C++.]

Be careful to use the right Visual C++ runtime DLL. In particular, Windows 95 ships with one version of the DLL in the '`\WINDOWS\SYSTEM`' directory, since many of the Windows 95 system applets are written with Visual C++.

10.1.2 Prerequisite software to use AND develop ILU applications on Windows NT and 95.

This release of ILU for Windows NT was originally developed with Microsoft Visual C++ Version 2.0, on Windows NT 3.5, and we now build on Windows NT 4.0 under Visual C++ 6.0. We have not tried with any other compiler. The ILU runtime DLLs for NT are 32 bit, and a 32 bit compiler is needed to develop applications that use them. If you succeed in building ILU or ILU applications for NT with a compiler other than Microsoft Visual C++, please report your findings. We simply haven't had time to test ILU with other C or C++ compilers with Windows.

Typically, we move to the latest version of Visual C++ as soon as it has been shipped to us. There is nothing known in ILU that should prevent it from being built with earlier versions of Visual C++.

10.2 Installation

ILU comes prebuilt for Windows NT. For the current release of ILU, a single '`.ZIP`' file is the prebuilt version. The '`.ZIP`' file is created with Nico Mak Computing's `WINZIP`, which allows long file names and is available for all versions of Windows. However, if you only have `PKZIP`, you should be able to extract the files from the '`.ZIP`' with no problems. Just make sure you use the **-d** when unzipping so that `PKZIP` will preserve the directory structure contained within the '`.ZIP`' file. You should also be aware that we use long filenames for some of the stubbers, so older versions of PKZIP might truncate the filenames as the files are extracted.

Determine where you wish to install ILU, e.g. '`C:\ILUWIN`'. Set the environment variable *ILUHOME* to this directory (*ILUHOME* is needed for building the examples). Unpack the distribution into your installation directory using `pkzip -d iluwin.zip` (or, if using `WINZIP`, just open the '`.ZIP`' archive and press the "Extract" button). You should now have subdirectories in *ILUHOME* called '`bin`', '`examples`', '`include`', '`interfaces`' and '`lib`'.

If you'll be developing ILU apps, or building the examples, set the environment variable *ILUPATH* to include '*ILUHOME*`\interfaces`' *ILUPATH* is the path of directories where interface ('`.isl`') files can be found. For example, setting *ILUPATH* to '`.;C:\ILUWIN\INTERFACES`' will cause ILU stubbers to look for interfaces first in the current directory, then in '`C:\ILUWIN\INTERFACES`'.

Add the '*ILUHOME*`\bin`' directory to your *PATH* environment variable.

Determine what common directory share will be used for your applications to publish information about ILU objects. This will commonly be a directory that is exported from a file server and shared by all the systems. Set the environment variable *ILU_BINDING_DIRECTORY* to this directory e.g. *ILU_BINDING_DIRECTORY*=`f:\iluwin\bindings`. If you do not set this, ILU will default to whatever value is specified in the file '`iluwin.h`'.

10.3 Building ILU

(For those who just *must* have and build the source! :-)

If you wish to build the ILU system from source, begin by obtaining the source distribution ('`ilu.tar.gz`'). There is no separate source tree for the Windows version; the same source code is used for both Unix and Windows. Set *ILUHOME* to where you will want ILU to be installed. Determine where you wish to install the ILU source, and set the environment variable *ILUSRC* to that directory e.g. '*ILUHOME*`\src`'. Unpack the distribution into that directory. Change to the *ILUSRC* directory. Having previously installed Visual C++, perform

```
> nmake -f iluwin32.mak
```

To subsequently install into *ILUHOME*, perform

```
> nmake -f iluwin32.mak install
```

Note that the default is to build a 'release' version. If you wish to build a 'debug' version perform

```
> nmake -f iluwin32.mak CFG="Win32 Debug"
```

To clean up after installation perform

```
> nmake -f iluwin32.mak clean
```

Various #defines that determine how ILU is built can be found in the file '`ILUSRC/runtime/kernel/iluwin.h`'.

When bulding the debug versions of the c, c++, and kernel runtimes, the values of the environment variables, *ILU_DEBUG_CFLAGS* and *ILU_DEBUG_CPPFLAGS* are passed to the c and c++ compiler command lines respectively. This allows the builder to do things like creating source browser files, e.g. set ILU_DEBUG_CFLAGS=/FR"/ilu/browsefiles/", set ILU_DEBUG_CPPFLAGS=/FR"/ilu/browsefiles/".

Note: "make clean" does not work across all versions of Windows. In particular, it will not work on systems other than Windows NT. If you are using Windows 95, just remove all occurrences of the '`WinDebug`', '`WinDebugW`', '`WinRel`', and '`WinRelW`' directories in the source tree and examples directories. You can also safely delete any .map, .ilk, .exp, and .pdb files you might see.

Note that it is normal to see a number of compiler warnings during the ILU build process. It's also been reported that linking in the build can fail if the full Visual C++ has not been installed. This is because the makefiles used were originally generated by the MSVC Development Environment, which by default adds a whole slew of libraries to the link command, e.g. odbc32.lib (even though ilu doesn't need them, apparently the link fails because they cannot be found). The workaround is to fully install Visual C++, or go modify the makefiles to take out those references.

10.3.1 Lisp

If you would like to use Allegro Lisp for Windows with ILU, you will have uncomment the appropriate lines in the '*ILUSRC*`/stubbers/lisp/iluwin32.mak`' and '*ILUSRC*`/runtime/lisp/iluwin32.mak`' makefiles, as these components are not built by default. Note that the Allegro Lisp for Windows support was graciously contributed by Joachim Achtzehnter and has not been tested at PARC. For more information, see `http://vanbc.wimsey.com/~joachim/ilu.html`.

10.3.2 Java

ILU's supports Java on Windows using either Javasoft's JDK 1.1 `http://www.javasoft.com/products/jdk/1.1/i` or JDK 1.2 `http://www.javasoft.com/products/jdk/1.2/index.html`, or Microsoft's SDK for Java 3.1 Release (Visual J++ 6 compatible)`http://www.microsoft.com/java/download.htm` . If you're using the Javasoft JDK,you should define the environment variable *JAVASDK* to Javasoft. If you're using Microsoft's SDK for Java 2.0 Beta 2 Release, you should define the environment variable *JAVASDK* to Microsoft. In both cases, the environment variable *JAVA_HOME* should be set to wherever the Java development software was installed, e.g. set JAVA_HOME=e:\jdk1.1.6, or set JAVA_HOME=e:\SDK-Java.20. Be sure your *CLASSPATH* environment variable includes the *ILUHOME*/lib/classes directory. e.g. set CLASSPATH=.:.\classes:e:\iluwin\lib\classes.

NOTE: In the Microsoft's SDK for Java 2.0 Beta 2 Release, you must modify the file *JAVA_HOME***include****native.h to correctly reflect the fact that the function RNIGetCompatibleVersion is something exported from a dll to Java. i.e.:**

```
#pragma message( "NOTE: ILU mod to native.h - Defining RNIGetCompatibleVer-
sion as dllexport" )
/* originally  DWORD __cdecl RNIGetCompatibleVersion(); */
__declspec(dllexport) DWORD __cdecl RNIGetCompatibleVersion();
```

```
NOTE: An ilu build upon Javasoft's JDK 1.1.6 can be also be used from
applications within JBuilder 2, or, Visual Cafe 2.5.
```

```
ILU's support for Javasoft's JDK 1.2 needs a few environment variables set
accordingly:  You need to set the environment variable JAVA_HOME to the
jre directory and the
JDK_HOME to the jdk directory.  Furthermore the mak file requires the en-
vironment variable JAVAUSE12 to be set.

ILU's also used to support Java using Javasoft's JDK 1.2beta2 or JDK 1.2beta4
pre-release, but the build may be more tricky and no more recommended because
our testing does not include outdated beta releases.
```

```
NOTE: Build a jar file and the javadoc files manually.
```

```
NOTE: It is possible to manually create a release directories containing
all
Javasoft's JDK 1.1.6 support, JDK 1.2 support and Microsoft's SDK support
in a friendly
co-existing way.  To do this with the current make files: build and install
for
Microsoft first.  Then in a new shell (setup to build for Javasoft's JDK)
you
can cd into the src\runtime\java directory and re-build just the java runtime.
Go back to the top and re-install ALL of ilu.  (Do it in this order, so that
the "better" files are installed later.  The three builds use non-conflicting
names for the one otherwise incompatible dll file; all other files are
compatible as long as you are not trying to single step or debug ilu it-
self).  The
build is not quite perfect: If building a further ilu system fails because
existing files from the first
```

```
build interact badly, clean ilu (having set the environment variables for
the later system) and build
ilu system again.  The clean step will leave the important file from the
first build alone.
```

10.3.3 Python

By default, the Python Language Specific Runtime is only built if the environment variable PYTHONSRC is set to point to your Python Source directory which contains the Python Include directory, the Python PC directory, and (for Python 1.4, the Python vc40 directory containing python14.lib) (for Python 1.5.1, the PCBuild\Release directory containing python15.lib).

If you need these files, please retrieve the Python source from http://www.python.org, as python1.4.tar.gz for Python 1.4, and pyth151.tgz for Python 1.5.1. They are NOT distributed with ILU.

Python 1.5.1 build is the default ILU Python runtime on Win32.

You will need to put ILUHOME/lib on your PYTHONPATH before using ILU with Python.

If you're going to build ILU from the source tree, and you want to use Python, build Python yourself first!

For Python 1.5.1, Ensure that you build the Python 'Release' Configuration as this is set up to use the Multithreaded C Runtime DLL. If you really want to build the 'Debug' version of Python, you must adjust the project to use the Multithreaded C Runtime DLL, NOT the Debug Multithreaded C Runtime DLL. Failure to do this will cause strange problems as ILU always uses Multithreaded C Runtime DLL. Also, if building the 'Debug' version of Python, you should adjust all references to python15.lib in the iluPr15.mak file to reflect the fact that it's in $(PYTHONSRC)\PCbuild\Debug

For Python 1.4, Edit the file in the PC directory called python_nt.def, so as to also export the following symbols: start_new_thread init_thread get_thread_ident exit_thread (and for compatibility with pythonwin, PyArg_ParseTupleAndKeywords). Then follow the instructions in the PC/readme.txt file to build Python. Next change the line for 'ALL' in $(ILUSRC)\runtime\python\iluwin32.mak to ALL : PMAKE14 before building ILU's Python runtime. NOTE: You *must* build Python itself. ILU requires several symbols to be exported from the python.dll that are normally not normally exported. Before building Python, in the Python source tree, edit the file in the PC directory called python_nt.def, so as to also export the following symbols: start_new_thread init_thread get_thread_ident exit_thread (and for compatibility with pythonwin, PyArg_ParseTupleAndKeywords). Then follow the instructions in the PC/readme.txt file to build Python. In addition to exporting these additional symbols, building Python yourself ensures that Python is using the same C runtime library as ILU. Different C runtime libraries in use at the same time will result in bizarre, hard-to-track-down behavior.

10.4 Building the examples

To build the examples, cd to '*ILUHOME*`\examples`'. Ensure that you have set *ILUPATH* as previously discussed.

For Windows examples, perform

```
> nmake -f iluwin32.mak
```

If you wish to build a 'debug' version perform

```
> nmake -f iluwin32.mak CFG="Win32 Debug"
```

This will create the example executables in subdirectories of the '`example`' subdirectories, called '`WinRel`' and '`WinRelW`' (or '`WinDebug`' and '`WinDebugW`' if you built a debug release) which correspond to the non-Windows and Windows versions of the examples.

Note that it is normal to see a number of compiler warnings during the examples build process.

10.5 Running the examples

Ensure that you have set *ILU_BINDING_DIRECTORY* as previously discussed. The non-Windows NT examples operate just like their Unix counterparts. The Windows examples are simple Windows versions of the same programs. To execute them, launch the executables (from the Windows File Manager, a command prompt (if you are running Windows NT or 95, or whatever), and choose the 'Run' entry from the 'Action' menu.

10.6 Developing Windows Applications with ILU

The basic process for using ILU in a Windows application is simple. You either write a new interface description or use an existing one. You run the stubbers against the interface description to generate stub code. You write calls to the methods exported from the interface in your application, or implement the object type in your application, depending on whether you're using the module, or providing it. Finally, you link your application code together with the generated stub code and the ILU libraries.

10.6.1 All Applications

Ensure that *WIN32* is defined to the preprocessor when building a 32 bit ILU application. This is normally set by default by Visual C++, but you should verify.

You need to link with the language specific runtime, the kernel runtime, and the winsock library.

Set the Visual C++ code generation compiler option to use the Multithreaded using DLL C runtime on Windows NT. This is *very* important. If you don't do this, then you'll run into a similar problem that was described above for the Python runtime. Essentially, if you create an application that doesn't use the Mutithreaded DLL runtime library, then the ILU kernel will be using one copy of the runtime library, and your application will be using a completely different one. This will cause all sorts of bizarre behavior. If you are debugging your application and you get all sorts of ASSERTs about memory allocation arenas, you've made this mistake.

There is NO need to call `ilu_StartWinsock` for a Windows NT ILU app. (It is taken care of for you internally in the runtime DLL process attach code).

10.6.2 Windows (non-console) Applications

We suggest you review and understand the test1 examples before you try to build a windowed ILU application. This section tries to highlight some of the *important* points. Admittedly, the Windows examples are simple and crude as Windows apps go, but they illustrate what you need to do in an application.

In C++ ILU apps, you'll be including '`Windows.h`'. However, '`Windows.h`' includes '`winspool.h`' and this file #defines `AddPort` as `AddPortA`. This interferes with `iluServer::AddPort()`, so you have to undefine it (temporarily at least). See the '`examples\test1\cppsrvrw.cpp`' file for an example.

10.6.3 Message Loop

See the windowed test1 server examples for a simplistic timer based means of using ILU in the presence of a Windows message loop. ('`msgalarm.c`'). You'll want to do something about the message loop since otherwise your Windows app won't service the GUI - it'll just be blocked in an internal call to `select()` waiting to deal with ILU activity. This simple timer approach makes use of the ability to associate an 'alarm' function with the ILU mainloop. When the alarm goes off (the example uses every 500 milliseconds), the alarm function processes any Windows messages that are waiting, then sets the alarm for another period.

Note that the test1 examples were developed with Microsoft's `TCP/IP` for Windows for Workgroups. Some of the behavior may be different under a different winsock implementation (especially with respect to message dispatch during `select()` calls). If so, please let us know.

10.7 Windows and the ILU_DEBUG settings

The "Debugging" chapter of this manual describes the facilities available to ILU developers for tracking down problems in their applications. One of these facilities is the *ILU_DEBUG* variable. When set to a value, it causes the ILU kernel and runtime to output various debugging messages.

For applications running under the Unix operating system, all output is sent to the stderr file handle, which can be redirected via the normal shell redirection operators. However, under Windows, this same flexibility is missing, since the Windows "shell" ('`CMD.EXE`' or '`COMMAND.COM`', depending on which version of Windows you are running) doesn't have the same flexbility. Programs that aren't console applications have an additional problem: they don't have any place for the output to go, since so-called "Windowed-API" applications detach themselves from consoles if they are executed from a command line.

The debug module in ILU has special code to handle this situation under Windows. Whether or not ILU outputs any messages depends on the "Debug Level" setting. This can be set two ways: either using the

ILU_DEBUG environment variable, or via the ilu_SetDebugLevel() function (see 'debug.c' for the whole story).

Normally, when the ILU kernel loads, it checks to see if *ILU_DEBUG* is set. If it is, it sets the appropriate debug level in the kernel, and then provides an internal error handler who's only job is to take the messages sent out by the kernel and write them to stderr. Of course, there is a function available to let you specify your own error handler. Just keep in mind that if you don't provide an error handler, and the kernel outputs a message because of *ILU_DEBUG* or ilu_SetDebugLevel() being called, then (by default), the debug messages will be sent to stderr.

If you are working with a console-based application under Windows, then this is not a problem. The messages will appear in the console that your application owns. Unfortunately, you can't redirect them to a file via the command line, since the Windows shell won't let you redirect arbitrary file handles. You can use *ILU_DEBUG_FILE* to redirect debug messages to a named file.

But, if you are working with a "real" Windows program, there is no console, and sending anything to stderr causes no output (since Windows equates stderr to the bit-bucket for Windowed-API applications). If you don't take any actions, then the kernel will handle this for you. If the debug level gets set (under Windows), the kernel attempts to figure out if your application is Windows-based or console-based. If it's console-based, then the normal debug-output functions are used.

However, if it appears that you are running a Windows-based application, and you have not provided an debug-handler of your own, then the kernel will create "Debug Console" and send all the debug output there for you.

This means that if you would like to set ILU_DEBUG, and your application is not console-based, then you don't have to do anything special to see the debug output. It's handled for you.

There are several things to keep in mind about this. The kernel has a very narrow-minded view of how to handle this. If you create your own debug handler, and you want to have the debug messages sent to you, *don't* set *ILU_DEBUG*. Instead, have your application set the debug level *after* you have installed your debug handler. If a debug handler other than the default gets installed, the code in 'debug.c' will assume that you are going to handle it automatically, and it won't set up a debug console.

The way to determine if your application is console-based or Windows-based is to attempt to create a console (if you know a better way to determin this, please pass it on). This is what the kernel does; if the call to the WIN32 API call AllocConsole() succeeds, the kernel assumes (rightly) that your application doesn't have a console, and thus is a Windowed-API application. But, if your app is a Windowed-API application, and you create your own console before the kernel does, then AllocConsole() will fail, and the kernel will use *your* console (which may not be what you want).

So, just keep this is mind: If your application is a Windows app, then setting ILU_DEBUG will work for you. Just remember that if you want to capture the debug output yourself, you have to make sure and set up your handler *before* the debug level gets set in the kernel. See the code in 'debug.c' for the whole story

10.7.1 WINIO

Note: ILU no longer needs WINIO. If the kernel or a runtime needs to output a message: if the application is a console app, output will go to that console; else the application must be a Windows app, and ILU will create a console window to which output will be sent.

WINIO is no longer distributed with ILU.

10.8 Misc.

This section contains a 'hodge-podge' of information - with little attention paid to formatting.

10.8.1 Python 1.4 support - details

This release of ILU supports both Python 1.4 Python 1.5.1 (the default) for Windows. If you want to use Python 1.4, building it as described in the previous Python section of this chapter is a bit involved. Specifically, here's what you need to do:

Assuming we don't have Python at all on the machine.

1. Retrieve the python source from http://www.python.org, as python1.4.tar.gz.

2. Edit the file in the PC directory called python_nt.def, so as to also export the following symbols: start_new_thread init_thread get_thread_ident exit_thread and for compatibility with pythonwin, PyArg_ParseTupleAndKeywords).

3. Then follow the instructions in the PC/readme.txt file to build python. (Ensure that the resulting python14.dll is the dll that gets used by python.)

4. Go to ILUSRC\runtime\python and comment out the line in the iluwin32.mak makefile to allow the build of the runtime.

5. Set the PYTHONSRC environment variable appropriately, e.g. set PYTHONSRC=E:\Python-1.4src

6. If you DON'T want thread support in Python, remove or comment out the line #define ILU_PYTHON_THREADS 1 from the file ILUSRC\runtime\python\pythonversion.win

8. Make ILU - you should now have the file iluPr.pyd in the appropriate build subdirectory of ILUSRC\runtime\python.

9. Make Install ilu - This will copy the *.py files in ILUSRC\runtime\python to ILUHOME\lib. Be sure to put \ILUHOME\lib on your on your PYTHONPATH.

10. Enjoy.

You can run several of the Python examples from ILU (the Python versions of Test1 and Bank work; Reconnect needs one change to work; change `"import socket"` to `"import _socket"`).

10.8.2 Alternative Binding Service

The ilu binding service (in ILUSRC/etc/sbserver) is not built under Windows. Basically, the steps to build it are:

Step 1: Modify the makefile for the ILU kernel to include sbilu.obj instead of sbfile.obj. If you are building the kernel from scratch and don't care about dependencies, just replace "sbfile" with "sbilu" everywhere you see it in ilu32.mak.

Step 2: Modify the iluwin.h configuration header file to set the necessary parameters for the service. Specifically, these are the lines you need to worry about:

/* Define this to be the value of the ILU simple binding directory, if using shared files for simple binding */ #if !defined(ILU_BINDING_DIRECTORY) #define ILU_BINDING_DIRECTORY "\\project\\rpc\\current\\lib\\binding" #endif

Make sure that ILU_BINDING_DIRECTORY is *not* set to a value anywhere in the system. As you can see from the text, it gets automatically set if you haven't assigned a value. Comment out these lines so that it doesn't get set. The system decides what type of binding to use based on ILU_BINDING_DIRECTORY having a value or not.

/* Define this to be the domain of the simple binding server, if using ILU service for simple binding */ /* #undef ILU_BINDING_REALM */

/* Define this to be the host ip addr of the simple binding server, if using ILU service for simple binding */ /* #undef ILU_BINDING_HOST */

/* Define this to be the network port on the binding host, if using ILU service for simple binding */ /* #undef ILU_BINDING_PORT */

You'll need to uncomment these #defines and give them the appropriate values. See iluconf.h for more information about these variables.

Once you've done all this, you can make the kernel and it will have the simple binding service enabled and working.

The /ilu/etc directory contains directories called sbfile and sbserver. Between the two of them, you can build the simple binding server for Windows.

10.8.3 Borland C

ILU source does not build with Borland C, but some successful attempts have been reported. Here are some of the hints that have been passed back.

Borland doesn't prepend an underscore on Unix-based lib functions - all the various calls in "runtime/kernel/os/win.h

Header files - doesn't like declarations of partially typed variables. e.g. just saying extern struct _ilu_DefaultAlarm_struct. This forces the definition of the entity in the header file, requiring it to be removed from the source file where it is currently fully defined.

.def files need to have the VERSION keyword removed, and .def files that reference functions need to be changed to reference those functions without underscores. Microsoft uses underscores, Borland doesn't.

Whatever method you use for building ILU, make sure and set the "Max errors and warnings" to 0 (don't stop). ILU generates a number of warnings when build with Borland, and not setting this will cause the build to fail with a "too many warnings" error.

Don't enable CodeGuard (Borland's C/C++ memory check library; it looks for leaks and other nasties and logs them). It will have a field day with ILU since some parts of the system leak on purpose, like the stubbers.

10.9 Files in the distribution

Note: this list is in the process of being updated, and may contain some errors.

bin directory

- '`c++-stubber.exe`' -- console C++ stubber
- '`c-stubber.exe`' -- console C stubber
- '`ilu32.dll`' -- kernel runtime DLL
- '`iluc32.dll`' -- C runtime DLL
- '`ilucpp32.dll`' -- C++ runtime DLL
- '`islscan.exe`' -- console ISL scanner
- '`idl2isl.exe`' -- IDL to ISL converter
- '`parser32.dll`' -- parser DLL for stubbers
- '`python-stubber.exe`' -- console python stubber

lib directory

(Note unlabeled entries are the import libraries for their counterparts in the bin directory)

- '`ilu.py`' -- python runtime file
- '`ilu32.lib`'
- '`iluPr.pyd`' -- a dll used by the python runtime
- '`iluRt.py`' -- python runtime file
- '`ilu_tk.py`' -- python tk file
- '`iluc32.lib`'
- '`ilucpp32.lib`'

- ‘parser32.lib’

include directory (header files need for building ILU apps)

- ‘ilu.hh’ -- main c++ runtime header file
- ‘ilubasic.h’
- ‘iluchdrs.h’ -- main c runtime header file
- ‘iluconf.h’
- ‘ilucstub.h’
- ‘iludebug.h’
- ‘iluerror.h’
- ‘iluerrs.h’
- ‘iluhash.h’
- ‘iluntrnl.h’
- ‘iluptype.h’
- ‘ilutypes.h’
- ‘iluxport.h’ -- kernel exports header file
- ‘iluprotocol.h’
- ‘ilutransport.h’

interfaces directory

- ‘ilu.isl’
- ‘http.isl’

examples directory

- ‘iluwin32.mak’

examples/httest

- ‘htclient.c’
- ‘htclient.mak’
- ‘htserver.c’
- ‘htserver.mak’
- ‘httest.isl’
- ‘iluwin32.mak’
- ‘README’
- ‘webserver.c’
- ‘webserver.mak’

examples/iioptest1

- ‘client.c’
- ‘client.mak’
- ‘iioptest1.isl’
- ‘iluwin32.mak’
- ‘server.c’
- ‘server.mak’
- ‘test1.idl’

examples/test1

- ‘clnconsl.ico’
- ‘clnt.c’
- ‘clnt.h’
- ‘clnt.mak’
- ‘clntu.c’
- ‘clntw.c’
- ‘clntw.ico’
- ‘clntw.mak’
- ‘clntw.rc’
- ‘cppclnt.cc’
- ‘cppclnt.cpp’
- ‘cppclnt.mak’
- ‘cppclntw.cpp’
- ‘cppclntw.mak’
- ‘cppsrvr.cc’
- ‘cppsrvr.cpp’
- ‘cppsrvr.mak’
- ‘cppsrvrw.cpp’
- ‘cppsrvrw.mak’
- ‘iluwin32.mak’
- ‘msgalarm.c’
- ‘pyClient.py’
- ‘pyServer.py’
- ‘README’

* ‘resource.h’
* ‘srvr.c’
* ‘srvr.h’
* ‘srvr.mak’
* ‘srvru.c’
* ‘srvrw.c’
* ‘srvrw.ico’
* ‘srvrw.mak’
* ‘srvrw.rc’
* ‘svrconsl.ico’
* ‘Test1.isl’
* ‘Test2.isl’
* ‘Test3.isl’

examples/test2

* ‘Batcher.isl’
* ‘bclient.mak’
* ‘bclnt.c’
* ‘bserver.mak’
* ‘bsvr.c’
* ‘Fibber.isl’
* ‘fibber.mak’
* ‘fprog.c’
* ‘iluwin32.mak’
* ‘README’

examples/timeit

* ‘client.c’
* ‘client.mak’
* ‘iluwin32.mak’
* ‘prebuiltISL’
* ‘README’
* ‘server.c’
* ‘server.mak’

11 Binding Names in ILU

11.1 Introduction

This chapter explains some of the issues around binding names to objects and services, and the various mechanisms that ILU provides to deal with them.

11.2 ILU Simple Binding

ILU includes a simple binding/naming facility. It allows a module to *publish* an object, so that another module can import that object knowing only its object ID (as defined in Chapter 1 [ILU Concepts], page 2). The interface to this facility is deliberately quite simple; one reason is to allow various implementations.

The interface consists of three operations: *Publish*, *Withdraw*, and *Lookup*. `Publish` takes one argument, an ILU object. `Publish` returns a string that is needed to successfully invoke `Withdraw`. `Withdraw` undoes the effects of `Publish`, and takes two arguments: (1) the object in question, and (2) the string returned from `Publish`. In some langauge mappings, the string is not explicitly passed, but conveyed in the language mapping's representation of ILU objects. `Lookup` takes two arguments: an object ID and a type the identified object should have. If the object with that ID is currently being published, and has the given type (among others), `Lookup` returns that object.

The implementation shipped with this release of ILU can use either an ILU service, or a shared filesystem directory, to store information on the currently published objects. This choice must be specified at system configuration time. If the shared filesystem approach is used, this directory must be available by the same name, on all machines which wish to interoperate. The way in which clients interact with binding is the same, regardless of which approach is selected.

11.2.1 Listing the Registered Objects

The simple program `ilusbls` will list the currently registered objects. It may be invoked with an argument, in which case only those objects with string binding handles containing the argument as a substring will be listed.

11.2.2 Using Shared Files for Simple Binding

If simple binding is to be done with shared files, a default directory is compiled into the ILU library. This directory may be explicitly specified at system configuration time with the `--with-binding-dir=`*DIRECTORY* switch to `configure`. (The compiled-in setting may also be overridden at run time, by setting the environment variable `ILU_BINDING_DIRECTORY` to a different directory.) ILU creates a file in this directory for each published object. The name of the file is an 8-digit hex string, formed by taking the CRC-32 hash of the server ID and instance handle of the object. The file contains the string binding handle of the object and a random string, which serves as the 'proof' that has to be provided when withdrawing a registration. Note that when using the shared file approach, the protection state of the directory must be such that programs calling `Publish` can remove files and create new files, and programs calling `Lookup` must be able to read files in the directory.

11.2.3 Using an ILU Service for Simple Binding

If an ILU service is used, the situation is a bit more flexible. The idea is that a program called `ilusb` is run on some machine, and exports the binding service via a specified port. All clients have compiled-in knowledge as to which machine and port the binding service is running on, and they contact the service to perform `Publish`, `Withdraw`, and `Lookup` calls. Each binding service is given a name, called a *binding realm*, which is the name of the 'space' for which it provides simple binding services. There may therefore be many instances of the `ilusb` server running, even on a single machine, each one serving a different binding realm. It is often useful to establish multiple binding realms for different purposes. For instance, one might be used for everyday registration of services, another for testing, still another for experimenting.

To start the binding service, run the program *ILUHOME*`/bin/ilusb`. It takes the following options:

- `-r` *REALM-NAME* -- this allows specification of the *REALM-NAME* which the server will serve. The default is the compiled-in realm name.
- `-h` *IP-ADDRESS* -- this allows overriding the compiled in IP address for the machine. This switch is mainly for multi-ported machines (machines which have two or more different IP addresses).
- `-p` *PORT* -- this allows overriding the compiled in *PORT* specification. This is the port on which the server listens for connections.
- `-f` *FILENAME* -- this allows you to specify the name of a file in which the server will store a backup of the various registrations currently active. If the server is restarted, and this file already exists, the server will read this file, and use the registrations in it as the current set of registrations. This file should be in a directory which can be read and written by the user identity under which the `ilusb` program is running. The default is a file called '`/tmp/ILUSimpleBindingService.`*REALM-NAME*'.
- `-s` -- this option enables 'protected' operation. This prevents active registrations from being overridden; `Withdraw` must be called first, to remove the current registration, before a new registration for the same object can be made. This is useful in an environment which only wants to permit certain users to change certain registrations. However, without secure communications, this operation is not truly secure, and enabling it is often clumsy for casual use of the simple binding service. The default is unprotected operation.

By default, ILU programs use the compiled-in binding realm, host, and port. However, they can be directed to use a different combination of these three, by defining the environment variable `ILU_BINDING_SERVICE` to a string of the form `"`*REALM*`:`*HOST*`:`*PORT*`"`, before running the program. If you want only to override one or two of the compiled-in defaults, use a string with empty fields for the other parts. For example, if you just wanted to redirect to a particular host, you could use a string of the form `":foo.bar.company.com:"`, with empty strings for *REALM* and *PORT*.

The program `ilusbls` will list all the currently registered objects. It takes an optional string argument. If the argument is specified, only objects which have that string in their string binding handles will be listed.

11.3 CORBA CosNaming

Another rather different notion of binding is covered by the CORBA standard ‘COS Naming’ service. An implementation of this service is installed with ILU if the configuration options include support for the C programming language, the OMG IDL interface definition language, and the `CORBA IIOP` wire protocol. The interface to the service is defined in ‘*ILUHOME*`/interfaces/CosNaming.idl`’.

To start the binding service, run the program *ILUHOME*`/bin/ILUCosNaming`. It takes the following options:

- `-p` *PORT* -- this allows specification of the port on which the server will listen for connection requests. If this is not specified, the server will use a randomly assigned free port, and the CORBA IOR for the service will differ from invocation to invocation. If you want the service to have an unchanging CORBA IOR or ILU SBH, this must be specified.
- `-h` *HOSTNAME* -- this allows overriding the default hostname for the machine. This switch is mainly for multi-ported machines (machines which have two or more different IP addresses). The default hostname is whatever is returned by the `gethostname()` system call on your machine.
- `-sid` *SID* -- this allows specification of a server ID for the root object exported by the naming service, and all subsequent naming contexts created in that server. If none is specified, the default used is `"CosNaming_`*HOSTNAME*`"`, where *HOSTNAME* is the hostname being used.
- `-ih` *IH* -- this allows specification of an instance handle for the root object exported by the naming service. This defaults to `"root"`.
- `-f` *FILENAME* -- this allows you to specify the name of a file in which the server will store a backup of the various registrations currently active. If the server is restarted, and this file already exists, the server will read this file, and use the registrations in it as the current set of registrations. This file should be in a directory which can be read and written by the user identity under which the `ILUCosNaming` program is running. The default is a file called ‘`/tmp/ILUCosNaming.`*SID*’, where *SID* is the server ID specified or defaulted to for the program.
- `-ior` -- if this option is specified, the service will write its CORBA IOR to the standard output after it has successfully initialized. This is sometimes useful when this service is being used with a non-ILU CORBA ORB.
- `-sid_and_ih_from_IOR` *STRINGIFIED-IOR* -- it is sometimes useful to use the ‘ILUCosNaming’ program to mimic the name service provided by another ORB. In particular, this means that the root object should have the same server ID and instance handle provided by that other service. To accomplish this, you can start `ILUCosNaming` with the stringified IOR for that other service, and it will use the server ID and instance handle in that IOR for its server ID and instance handle. Note that you may also need to specify the port and hostname with the `-h` and `-p` options to exactly mimic the other service.

In general, the user shouldn't have to specify the hostname, server ID, instance handle, or filename; the defaults work pretty well. It is useful to specify the port in most applications, just so that the service always has the same string binding handle, which will be
`ilu:`*SID*`/`*IH*`;IDL%3Aomg.org%2FCosNaming%2FNamingContext%3A1.0;iiop_1_0_1@tcp_`*HOST*_*PORT*
where *SID*, *IH*, *HOST*, and *PORT* are the values for the server ID, instance handle, hostname, and port, as given above.

You should look at any CORBA book, or the COS Naming service specification (`ftp://ftp.omg.org/pub/docs/formal/97-07-04.pdf`) itself, for more information on how to use the naming service with applications. Note that the ILU implementation does not include the client-side 'Names Library' described in the specification. Applications are expected to use the service functionality directly.

12 Threads and Event Loops

ILU can be used in either the single-threaded or the multi-threaded programming style. This chapter describes the relevant issues.

The issue of *threadedness* appears at two levels: within a program instance, and again for an entire distributed system. We will first discuss the program level, and then the system level.

Some programming languages are defined to support multiple threads of control. Java is an example. Other language definitions are single-threaded, or are silent on this issue. Some of these, such as C and C++, can be used to write mutli-threaded programs with the use of certain libraries, coding practices, and compilation switches. ILU can be used in multi-threaded program instances in both inherently multi-threaded languages and some of those where multi-threading is an option; similarly, ILU can be used in single-threaded program instances in both inherently single-threaded languages and all of those where multi-threading is optional.

12.1 Multi-Threaded Programs

Use of ILU in multi-threaded program instances mainly raises three issues: (1) ILU's use of thread resources, (2) how to switch ILU to multi-threaded operation (for languages where multi-threading is optional), and (3) thread synchronization issues. We'll take them in that order.

When threads are used, ILU will fork: (a) a few (currently 6, I think) threads at startup; (b) a thread for each port added to a true server, for each connection to that port, and --- if the RPC protocol for that port is concurrent (see See Section 13.2.5 [Protocols and Transports], page 216) and the language being used is one of C, C++, or Java --- a thread for each call received on that connection; and (c) a thread per user-level alarm (ref ???) (of which there are none by default).

ILU's runtimes for both Franz Common Lisp and Java support multi-threading; programmers do not need to do anything special in these languages.

ILU's runtimes for C, Python and C++ support both single-threaded and multi-threaded programming; they assume single-threading by default, and can be switched to multi-threading during initialization (described below).

12.1.1 Multi-Threaded Programming in C

By default, the ANSI C language support in ILU is single-threaded. However, an application can switch the ILU runtime kernel and ANSI C language support to multi-threaded operation; this involves informing the ILU runtime of which implementations to use for the threading primitives it needs. An application can directly supply any implementation it wants. To make this easy in certain common cases, there are predefined implementations available for use on systems that support POSIX, Solaris 2, or Win32 threads.

To directly supply the needed primitives, an application directly switches the ILU runtime kernel to multi-threaded operation (as described in Section A.4.1 [Control Structure Options], page 278) and furthermore gives ILU's C runtime a procedure that forks a thread (via `ILU_C_SetFork`, described in '`iluchdrs.h`').

Instead of directly supplying the needed primitives, an application running on one of the relevant systems can switch to multi-threaded operation by this macro invocation:

```
ILU_C_USE_OS_THREADS;
```

This can only be done if the option to do so was enabled during the configuration step of the installation of ILU; the default in that step is to enable this feature on systems that have an appropriate threading API, and not on others.

In either case, switching from single-threaded to multi-threaded operation must be done before any calls to `ILU_C_Run`, `ILU_C_InitializeServer`, or anything that relies on a default `ilu_Server` existing.

In some thread systems, it is important for the ''main'' thread not to exit before the program is finished executing. To provide for this, your C program should call `ILU_C_FINISH_MAIN_THREAD(`*val*`)` instead of simply returning from `main()`. This routine will block if necessary until it is safe for the thread to return, and will return the value *val*.

12.1.2 Multi-Threaded Programming in Python

If you have selected support for operating-system level threads and for Python in building ILU, and if you have installed Python with support for threading, the ILU support for Python will also support threaded operation, using the normal Python threads mechanisms.

By default, Python usage will be single-threaded. To switch the ILU Python runtime from its default assumption of single-threadedness to multi-threaded operation, call the Python function `ilu.ThreadedOperation()` before calling any ILU functions. This will switch both the ILU kernel and the Python runtime to multi-threaded operation.

12.1.3 Multi-Threaded Programming in C++

To switch the ILU C++ runtime from its default assumption of single-threadedness to multi-threaded operation, call `iluServer::SetFork` (described in '*ILUHOME*`/include/ilu.hh`') before calling `iluServer::Run`, `iluServer::Stoppable_Run`, `iluServer::iluServer`, or anything that relies on a default `iluServer` existing. `iluServer::SetFork` makes a feeble attempt to detect being called too late, returning a logical value indicating whether an error was detected (when an error is detected, the switch is not made). This detection is not reliable --- the caller should take responsibility for getting this right.

Pass to `iluServer::SetFork` a procedure for forking a new thread. This forking procedure is given two arguments: a procedure of one pointer (`void *`) argument and a pointer value; the forked thread should invoke that procedure on that value, terminating when the procedure returns.

Before calling `iluServer::SetFork`, you must switch ILU's runtime kernel to multi-threaded operation by calling `ilu_SetWaitTech`, `ilu_SetMainLoop`, and `ilu_SetLockTech` as mentioned later (see Section A.4.1 [Control Structure Options], page 278). ILU's C++ runtime takes care of forking the thread to call `ilu_OtherNewConnection`; you should not call `ilu_NewConnectionGetterForked`.

12.1.4 Thread Synchronization

Thread synchronization issues are almost invisible in the interfaces that application programers use. In multi-threaded operation, ILU objects may be operated on concurrently: a multi-threaded client can make concurrent calls, and a multi-threaded server may receive concurrent calls. ILU itself does no particular synchronization of application-level code --- that's left up to the application.

The one exception, already noted, is Section 1.3.1.8 [Object Tables], page 8; their operation is inside certain of the ILU runtime's mutexes. For the sake of these (and any unexpected place this issue shows up), we now explain thread synchronization in ILU.

ILU uses mutex and condition variables to organize its thread synchronization; mutexes are even used for the plan for how single-threaded code works. A *mutex* is a data item that stands for a mutual exclusion condition. A mutex can be *held* by at most one thread at a time. We say a thread is either *inside* a mutex (when it holds the mutex) or *outside* a mutex (when it doesn't hold the mutex). Thus a region of code in which at most one thread may be executing at a time is surrounded by mutex *acquire* (aka *entry*) and *release* (aka *exit*) operations. The operation of entering a mutex blocks the calling thread until no thread is inside the mutex, then enters.

One particularly principled way of thinking about how to use mutexes involves associating a *mutex invariant* with a mutex; the invariant is expected to hold, except perhaps while the mutex is held. Just after the mutex is acquired, the invariant is known to hold; it will continue to hold until the code holding the mutex changes variables involved in the invariant; such code must restore the invariant before releasing the mutex. Note that this can be a useful way to understand the operation of even single-threaded code (particularly code with a recursive main loop); for this reason ILU code is fully annotated with respect to mutexes, even though that code may operate single-threaded: when a single-threaded program tries to acquire a mutex it already holds, a bug is revealed that would otherwise be much harder to track down. We do not have a comprehensive description of the full invariant associated with each ILU mutex, but we do have parts of some written down near the declarations of the variables involved (in particular, the comments at a variable's declaration may be taken as part of the invariant associated with the mutex that is the variable's lock (see below)).

Another way to think about mutexes is to use them as *object locks* (this is a technical term you will find in the literature; it has nothing to do with `"objects"` as in OOP, but uses a broader sense of the word). A shared (among threads) variable has a controlling lock, which must be held while reading or writing the variable. ILU rigorously includes *locking comments* that take the object lock view: for each variable the corresponding lock (or locks, in some cases) is indicated, and each procedure or method has declared pre- and post-conditions describing which locks must or must not be held.

ILU has two classes of mutexes: connection mutexes and non-connection mutexes. While connections don't show up in the ILU API, they do appear internally, and the connection mutexes figure into the Main Invariant, which applies to most application code. We will give no details on connection mutexes in this section, because applications don't manipulate them.

Here are the non-connection mutexes:

```
smu: global mutex for the server table;
otmu: global mutex for object type data structures;
cmu: global LRU list of connections
prmu: global mutex for protocol registry
trmu: global mutex for transport registry
gcmu: global mutex for GC data structures
timu: global mutex for alarm implementation.
server: one mutex per server.
```

Our main technique for avoiding deadlocks is to put a partial order on mutexes, and acquire mutexes in an order consistent with the partial order. That is, a thread may enter mutex B while holding mutex A only if A < B (we have a few carefully managed exceptions to this rule, involving connection mutexes). For non-connection mutexes, the partial order is the transitive closure of the following relationships:

```
cmu < server
smu < server
server < prmu
server < trmu
gcmu < server
gcmu < timu
cmu < smu
gcmu < cmu
cmu < timu
prmu < otmu
```

We use the symbols *L2* and *L1* to stand for the sets of connection and non-connection mutexes held by a thread, respectively. We write ">=" for the set inclusion relation. We write "L1.sup < X" to mean that either (a) L1 is empty, or (b) the maximum elment of L1 (the partial order rule says there must be exactly one maximal element whenever L1 isn't empty) precedes X in the partial order. We write "L1.sup = X" to mean that L1 is not empty and its maximum member is X. We don't speak of "L2.sup" because a thread is allowed to violate the partial order rule with respect to L2 mutexes.

There is a locking invariant called the *Main Remnant*, but it is only about connection mutexes.

There is a common locking invariant, called the *Main Invariant*:

```
L1 = {} and Main Remnant.
```

It holds in many places. The Main Invariant is exactly what's guaranteed to hold while an application's service routines are called. The Main Invariant is among the things guaranteed to hold while a stub is marshalling or unmarshalling. The Main Invariant is exactly what's guaranteed to hold while waiting for

I/O on a File Descriptor to be enabled. The Main Invariant is among the things guaranteed to hold while doing I/O on a File Descriptor.

For variables, the locking comments say what mutexes must be held to access the variable. For procedure values, the locking comments say what mutexes must be held to call the procedure, and, if the procedure changes the set of held mutexes, how. Both sorts of comment are applicable to procedure-valued variables; we prefer to document the locking pre- and post-conditions in a typedef of the procedure type, and describe the variable/mutex association in the usual way.

We have two sorts of locking comments: those about L1, and those about L2. Locking comments come in blocks. There are two kinds of blocks of locking comments: a `"sticky"` block is followed by a blank line; a `"one-shot"` is not. A locking comment is also called `"sticky"` or `"one-shot"`, depending on the kind of the comment block in which the comment is contained. A one-shot comment applies only to the immediately following item. A sticky comment of a certain sort applies to all items between it and the next sticky comment of the same sort, except those items to which a one-shot comment of the same sort applies.

The implementation of mutexes can itself be broken; when this is detected, an ILU procedure may raise an exception indicating this condition --- and when it does, the locking post-condition can not be expected to hold.

ILU also uses *condition variables* to get high-performance multithreaded operation. A thread can *wait* on a condition variable. Another thread can *notify* that condition variable. This causes all threads currently waiting on the condition variable to return from the wait operation. To prevent timing splinters, decisions about waiting and notifying should be made inside a mutex. This means the mutex must be released while waiting on a condition variable, and there must be no possibilty of a thread switch between the release of the mutex and the start of the wait; the wait operation thus takes the mutex as an argument, because in a pre-emptive threads environment the release and the wait must be an atomic thread operation.

12.2 Single-Threaded Programs

Users of ILU in single-threaded programs typically need to worry about only one thing: the main loop. To animate ILU server modules, a single-threaded program needs to be running the ILU main loop. This can be done, e.g., by calling `ILU_C_Run()` in C or `iluServer::Run` in C++. ILU also runs its main loop while waiting for I/O involved in RPC (so that incoming calls may be serviced while waiting for a reply to an outgoing call; for more on this, see Section 12.3 [Threadedness in Distributed Systems], page 197).

The problem is, many other subsystems also have or need their own main loop. Windowing toolkits are a prime example. When a programmer wants to create a single-threaded program that uses both ILU and another *main looped* subsystem, one main loop must be made to serve both (or all) subsystems. From ILU's point of view, there are two approaches doing this: (1) use ILU's default main loop, or (2) use some *external* (to ILU) main loop (this might be the main loop of some other subsystem, or a main loop synthesized specifically for the program at hand). ILU supports both approaches. Actually, ILU's runtime kernel supports both approaches. Currently no language veneers mention it. This is, in part, because it has no interaction with the jobs of the language veneers --- application code can call this part of the kernel directly (from any language that supports calling C code).

12.2.1 ILU Main Loop Functional Spec

ILU needs a main loop that repeatedly waits for I/O being enabled on file descriptors (a UNIX term) and/or certain times arriving, and invokes given procedures when the awaited events happen. (Receipt of certain UNIX signals should probably be added to the kinds of things that can be awaited.) The main loop can be recursively invoked by these given procedures (see Section 12.3 [Threadedness in Distributed Systems], page 197 for a good reason why), and thus particular instances of the main loop can be caused to terminate as soon as the currently executing given procedure returns.

For a detailed presentation, see the kernel interface version of this functionality, in procedures `ilu_RunMainLoop`, ... `ilu_UnsetAlarm` in '`iluxport.h`'; these are generic procedures that call the actual procedures of whatever main loop is really being used. To see the signatures of the procedures really being used, see the definition of type `ilu_MainLoop` in '`iluxport.h`'. [Should have a description here that doesn't reference '`iluxport.h`'.]

12.2.2 Using ILU's Default Main Loop

In this approach, ILU's default main loop is made to serve the needs of both ILU and the other main-loop-using parts of the program. When the other main-loop-using parts of the program need to register I/O handlers or use alarms, you arrange to call the appropriate generic procedures of the ILU main loop.

12.2.3 Using an External Main Loop

In this approach, you use an external (to ILU) main loop to serve the needs of ILU (as well as other parts of your program). This involves getting ILU to reveal to you its needs for waiting on I/O and time passage, and your arranging to satisfy these needs using the services of the external main loop. You do this by supplying to ILU, early in the initialization sequence, a metaobject of your creation. ILU reveals its needs to you by calls on the methods of this metaobject, and you satisfy them in your implementations of these methods. The C type for such a metaobject is `ilu_MainLoop`, found in '`iluxport.h`'.

Note that an `ilu_MainLoop` is responsible for managing multiple alarms. Some external main loops may directly support only one alarm. Later in '`iluxport.h`' you will find a general alarm multiplexing facility, which may come in handy in such situations.

See the files in '*ILUSRC*`/runtime/mainloop/`' for several examples of this approach (for the X Window System's various toolkits, like Motif, Xaw, XView, and Tk).

12.2.4 A Hybrid Aproach

Both of the above approaches rely on there being a certain amount of harmony between the functional requirements made by some main-looped subsystems and the functional capabilities offered by others. It also relies on the subsystems whose "normal" main loops are not used being open enough that you can determine their main loop needs. The conditions cannot be guaranteed in general. We've tried to minimize the main loop requirements of ILU, and maximize its openness.

We know of an example where neither of the above approaches is workable, and have a solution that may be of interest. See '*ILUSRC*`/etc/xview/`' for the (untested) code.

The problem is with the Xview toolkit (for the X Window System). Its main loop cannot be recursively invoked (a requirement of ILU), and the Xview toolkit is not open enough to enable use of any other main loop.

Our solution is to use Xview's main loop as the *top level* main loop, letting ILU use its own main loop when waiting on RPC I/O. Like the external main loop approach, this requires getting ILU to reveal its needs for waiting on I/O and time; unlike the external main loop approach, this requires *not* calling `ilu_SetMainLoop`. Instead of calling `ilu_SetMainLoop`, you call `ilu_AddRegisterersToDefault`, which causes ILU's default main loop to reveal ILU's needs to you --- in addition to doing everything the default main loop normally does. (Actually, the multiple alarms of ILU have been multiplexed into one here for your convenience.) You register these needs with the Xview main loop, and run it at the top level.

This solution is not as good as we'd like; it does not provide a truly integrated main loop. In particular, any I/O handler registered through ILU's generic procedures (`ilu_RegisterInputSource`, `ilu_RegisterOutputSource`) may be called spuriously: due to lack of coordination, both loops may decide a call is in order (when, of course, only one call is in order). As of release 2.0, ILU's own I/O handlers are prepared for spurious calls. Application programmers are responsible, when they use `ilu_AddRegisterersToDefault`, for making sure their I/O handlers that are registered through ILU's generic procedures are prepared for spurious calls.

12.3 Threadedness in Distributed Systems

In a distributed system of interacting program instances, you can (in principle, even if not (easily) in practice) trace a thread of control across remote procedure calls. Thus a distributed system, when viewed as a whole, can be seen to be programmed in either a single-threaded or multi-threaded style. ILU aims to minimize the consequences of the choice between in-memory and RPC binding, and this requires things not usually offered by other RPC systems. Some of these things are required by both the single-threaded and multi-threaded styles of programming distributed systems, for related but not quite identical reasons.

Forget RPC for a moment, and consider a single-threaded program instance. Method `m1` of object `o1` (we'll write this as `o1.m1`) may call `o2.m2`, which may call `o3.m3`, which may in turn call `o1.m1` again, which could then call `o3.m4`, and then everything could return (in LIFO order, of course). Late in this scenario, the call stack of the one thread includes two activations of the very same method of the same object (`o1.m1`), and another two activations of different methods of a common object (`o3`). All this is irrespective of module boundaries.

We want to be able to do the same thing in a distributed setting, where, e.g., each true object is in a different program instance. This means that while the ILU runtime is waiting for the reply of an RPC, it must be willing to service incoming calls. This is why ILU requires a recursive main loop in single-threaded programs.

In fact, one rarely wants single-threaded distributed systems. Indeed, the opportunities for concurrency are one of the main attractions of distributed systems. In particular, people often try to build multi-threaded distributed systems out of single-threaded program instances. While we hope this confused approach will fade as multi-threading support becomes more widespread, we recognize that it is currently an important customer requirement. Making single-threaded ILU willing to recursively invoke its main loop also makes single-threaded program instances more useful in a multi-threaded distributed system (but what you really want are multi-threaded program instances).

Threading is also an issue in RPC protocols. Some allow at most one outstanding call per connection. When using one of these, ILU is willing to use multiple parallel RPC connections, because they’re needed to make nested calls on the same server.

13 Protocols and Transports

When two modules of a program are in different address spaces, or use different data representations, ILU forms messages to send across the inter-module boundary; we call a particular way of forming and interpreting these messages an *RPC protocol* (sometimes simply *protocol*). These messages may be transported between address spaces in different ways; we call a particular way of moving messages a *transport*. This chapter describes the various kinds of available ILU protocols and transports. ILU is extensible: additional RPC protocols and transports can be added, either at compile-time or run-time; this chapter does not describe how to do so.

When an ILU kernel server exports objects, it does so via one or more contact stacks. Each stack has an RPC protocol at the top of the stack, forming and interpreting messages, and one or more layers of transport below the protocol layer, transforming or communicating the messages in various ways. A contact stack is specified by a protocol-info string and a sequence of transport-info strings; the syntax of these strings is defined in this chapter.

13.1 Protocols

13.1.1 The Abstract ILU Message Protocol

Before describing any particular protocol, we will describe the abstract ILU protocol, which is layered on top of each actual protocol. It is quite simple. Two types of messages are used, one to communicate parameters to a true method, and the other to communicate results and/or exceptions from the true method to surrogate caller. Parameters and values are encoded according to a simple abstract external data representation format. This abstract protocol identifies what information is passed between modules without specifying its exact mapping to bit patterns.

Additionally, the abstract protocol is capable of being used with either a *concurrent* or a *non-concurrent* actual protocol. A concurrent protocol is one which allows multiple requests to be outstanding on a single connection. A non-concurrent protocol normally allows only one outstanding request per connection; it preserves liveness with such protocols by using multiple connections when needed. This is to avoid introducing deadlocks into a distributed thread; ILU does not attempt any scheme for global thread identification. However, a client application can inform ILU that a collection of calls belong to logically distinct threads, thus enabling ILU to *pipeline* them down a single connection (if they happen to all be going to the same server). This is done by creating a pipeline meta-object and associating it with each of these calls.

ILU normally does not guarantee to deliver calls to server application code in the same order that client application code makes them (of course, a client can refrain from making one call until after receiving results from a previous one; this is not an instance of ILU doing anything to preserve ordering). However, a client can request special treatment of a collection of calls. That special treatment is an instance of the serialization guarantee. The guarantee is this: these calls are delivered to server application code in the same order that client application code makes them, except that a call made after a *barrier call* may be delivered before a call made earlier. A barrier call is one that raises the system exception `BARRIER`. No two of the given calls may be executed concurrently. The given calls must all be on objects in the same server. That server must export itself via a port that uses a non-concurrent protocol; that port must be the server's default port (the only one that clients attempt to use, so far). The serialization guarantee is requested on a collection of calls by creating a serializer meta-object and associating it with each of those calls.

13.1.1.1 Messages

The first type of message is called a *request*. Each request consists of a code identifying the method being requested, an authentication block identifying the principal making the call, and a list of parameter inputs to the method being called. The method is identified by passing the one-based ordinal value (that is, the index of the method in the list of methods, beginning with one) of the method, in the list of methods as specified in the ISL description of the class which actually defines the method. No more than 65278 (1-0xFEFF) methods may be directly specified for any type (though more methods may be inherited by a type). Method codes 0xFF00 to 0xFFFF are reserved for ILU internal use. The principal is identified by a block of authentication credentials information which varies depending on the specific authentication protocol used. These credentials may be either in the request header, or may appear as a parameter of the request. (Note: There should also be an ILU protocol version number somewhere here, but there isn't (yet).)

The *result* message is used to convey return values and exception values from the true method back to the caller. It consists of a Boolean value, indicating whether the call was successful (for `TRUE`) or signalled an exception (for `FALSE`). If successful, the return value (if any), follows, followed by the values of any `Out` parameters, in the order they are specified as parameters. If an exception was signalled, a value between 1 and 2^16-1 follows, indicating the ordinal value specific exception in the list specified in the definition of the method, followed by a value of the exception type, if any was specified for the exception.

13.1.1.2 Parameter Types

Simple numerical values, of types `integer`, `cardinal`, `real`, or `byte`, are passed directly.

`Character` values are passed as integer values in the range [0,2^16-1]. `Short character` values are passed as integer values in the range [0,2^16-1]. `Long character` values are passed as integer values in the range [0,2^32-1].

`Enumeration` values are passed as integer values in the range [0,2^16-1], the value being the zero-based ordinal value of the corresponding enumeration value in the original list of enumeration values in the definition of the enumerated type.

`Boolean` values are passed as as integer values of either 0, for `FALSE`, or 1, for `TRUE`.

`Optional` values are passed by first passing a Boolean value, with `TRUE` indicating that a non-`NIL` value is being passed, and then only in the non-`NIL` case passing a value of the optional value's indicated type.

`Sequence` values are passed by first passing a count, as an integer in the range [0,2^32-1] for sequences without limits, or for sequences with limits greater than 2^16-1, or an integer in the range [0,2^16-1], for sequences with limits less than 2^16, indicating the number of elements in the sequence, and then that number of values of the sequence's base type.

`Array` values are passed by passing a number of elements of the array's base type corresponding to the size of the array.

`Record` values are passed by passing values of types corresponding to the fields of the record, following the order in which the fields are defined in the ISL definition of the record.

`Union` values are passed by passing a value of the discriminant type, which indicates which branch of the union constitutes the union's actual type, usually followed by a value of the union's actual type. If the discriminant value indicates a branch of the union which has no associate value, only the discriminant value is passed.

`Object` values are passed in several different forms, depending on whether or not the object value is in the discriminator position, whether or not the object's type is a `singleton` type, and whether or not the object reference is `NIL`.

1. The first form is used when the object is in the discriminator position (that is, is the instance upon which the method is being invoked), and is an instance of a `singleton` type. In this case, the object is already known to both sides, and the object is passed implicitly; that is, no actual bytes are transmitted.
2. The second form is used when the object is in the discriminator position, but is not of a `singleton` type. In this case, the CRC-32 of the server ID of the object is passed as a `cardinal` value, followed by the instance handle of the object, as a `sequence of short character` value. Both the instance handle and server ID must be passed, as the true object previously at the ''known address'' for the object may have been replaced by a different object with the same instance handle, in a different kernel server.
3. In the third case, the object is being passed as a normal parameter, that is, not in the discriminator position. In this case, the full string binding handle of the object is passed as a `sequence of short character` value.
4. If the object being passed as a normal parameter is the CORBA Nil object reference, it is passed as the `sequence of short character` value of length zero.

13.1.2 The ONC RPC Protocol

This section describes the mapping of the abstract ILU protocol into the specific on-the-wire protocol used with ONC RPC[1] One of the major goals of this mapping is to preserve compatibility with existing Sun RPC services that can be described in ISL.

---

[1] RPC: Remote Procedure Call Protocol Specification, Version 2; R. Srinivasan. IETF RFC 1831, August 1995.

Four variants of the ONC RPC protocol are provided. They all have protocol info strings of the form *X*`sunrpc_2_`*program-number_program-version*, where *X* is either 'c' or nil, depending on whether the protocol is concurrent or not, respectively. *program-number* and *program-version* may be specified either in decimal, or in hexidecimal with a leading string of `0x`. The program number for non-native-ONC-RPC ILU object types is always the same (in ILU 2.0alpha1 to 2.0alpha7, `0x61A78`; in ILU 2.0alpha8 and up, `0x61A79`), and the program version varies depending on the specific object type.

Use of ONC RPC requires use of a boundaried transport below it.

13.1.2.1 Message Mappings

The request message used is that specified by the ONC RPC protocol. The ILU method index is encoded as a 32-bit number in the ''proc'' field in the ONC RPC request header. Principal identification is passed in the ''cred'' field of the ONC RPC request header. By default, ILU will pass the `AUTH_UNIX` authentication information, if no authentication method is specified for the method. (This default authentication can be disabled by defining the environment variable `ILU_NO_SUNRPC_UNIX_AUTH` to any value.) For non-singleton object types, the ONC RPC program number passed in the ''prog'' slot is always the same (for ILU 2.0alpha1 - 2.0alpha7, 0x00061a78; for ILU 2.0alpha8, 0x00061a79), and the version number passed in the ''vers'' slot is the CRC-32 hashed value of the MSTID for the object type on which the method being invoked is defined. For singleton classes, the program number and version specified in the singleton information is used. The ''mtype'' field is set to `CALL`. The indicated ''rpcvers'' is 2. A monotonically increasing 32-bit serial number is used in the ''xid'' field. For non-singleton, non-NIL objects, an extra argument identifying the *discriminant* of the message (the object on which the method is being invoked) is marshalled before any of the specified arguments. This discriminant is marshalled as an XDR Unsigned Integer, which is the CRC-32 of the server ID of the object, followed by an XDR string, which is the plain instance handle of the object.

The reply message used is that specified by the ONC RPC protocol. The ''mtype'' field is set to `REPLY`. The ''stat'' field is always set to `MSG_ACCEPTED`. In the `accepted_reply`, the authentication verifier is always NULL. The ''stat'' field may be non-zero, to signal one of a small number of ''standard'' exceptions, or may be zero. This header is then followed by one of three forms: If a ''standard'' exception was raised, nothing. If the method has no exceptions, the return values and out parameters (if any). If the method has any exceptions defined, a 32-bit value which specifies either successful completion (a value of 0), or an exception (a value greater than 0, which is the ordinal value of the particular exception being signalled in the list of exceptions specified for this method), followed by either the return value and out parameters (if any), in the case of successful completion, or the exception value (if any), in the case of an exception.

13.1.2.2 Mapping of Standard Types

The mapping of ILU types into ONC RPC types is accomplished primarily by using the appropriate XDR[2] representation for that type.

`Short integer` and `integer` types are represented with the XDR Integer type. `Long integer` types are represented as an XDR Hyper Integer.

`Short cardinal`, `byte`, and `cardinal` types are represented with the XDR Unsigned Integer type. `Long cardinal` types are represented as an XDR Unsigned Hyper Integer.

`Short real` numbers are encoded as XDR Floating-point. `Real` numbers are encoded as XDR Double-precision Floating-point. `Long real` numbers are encoded as XDR Fixed-length Opaque data of length 16.

`Array` values are encoded as XDR Fixed-length Array, except for two special cases. If the array is multi-dimensional, it is encoded as a flat rendering into a single-dimensional array in row-major order (the last specified index varying most rapidly). If the array is of element-type `byte` or `short character`, it is encoded as an array of one (in the one-dimensional case) or more (in the greater-than-one dimensional case) values of XDR Opaque Data.

`Record` values are encoded as XDR Structures.

`Union` values are encoded as XDR Discriminated Unions, with a discriminant of type ''unsigned int'' containing the ILU `short cardinal` discriminant.

`Enumeration` values are encoded as XDR Unsigned Integer (note that this is different from XDR Enumerations, which are encoded as XDR Integer).

`Boolean` values are encoded as XDR Unsigned Integer, using the value 0 for `FALSE` and the value 1 for `TRUE`.

`Sequence` values are encoded as XDR Variable-length Arrays, except for several special cases. Sequences of `short character` are encoded as XDR String, sequences of `byte` are encoded as XDR Variable-length Opaque Data, and sequences of `character` are encoded as XDR String, where the string is the UTF-2 encoding of the Unicode characters in the sequence.

`Optional` values are encoded as an XDR Boolean value, followed by another encoded value, if the Boolean value is TRUE.

Instances of an `object` type are encoded as either zero (in the case of a method discriminant of a singleton type), or one, values of type XDR String.

13.1.3 The Xerox Courier Protocol

This section describes the mapping of the abstract ILU protocol into the specific on-the-wire protocol used with Xerox Courier[3]. One of the major goals of this mapping is to preserve compatibility with existing

[2] XDR: External Data Representation Standard; R. Srinivasan. IETF RFC 1832, August 1995.

[3] Courier: The Remote Procedure Call Protocol; Xerox Corporation, XNSS 038112, 1981

Xerox Courier services that can be described in ISL. Unfortunately, many if not most important Courier services use *bulk data transfer*, something that is still only planned for ILU.

A protocol info string for Xerox Courier has the form `courier_`*program-number*_*program-version* where *program-number* and *program-version* may be specified either in decimal, or in hexidecimal with a leading string of `0x`. The program number for non-singleton ILU object types is always (in ILU 2.0) `0x001yxxxx`, where *y* is currently `1`; the specific program number and the program version varies depending on the specific object type. Courier is a non-concurrent protocol.

Use of Xerox Courier requires use of a boundaried transport below it.

13.1.3.1 Message Mappings -- Courier Layer 3

The request message used is the `CallMessageBody` specified in section 4.3.1 of the Courier protocol. A monotonically increasing 16-bit serial number is passed in the `transactionID` field; a 32-bit program number is passed in the `programNumber` field, a 16-bit number is passed in the `versionNumber` field; the ILU method index is passed as a 16-bit value in the `procedureValue` field. The program number is calculated by computing the CRC-32 hash value of the MSTID of the object type on which the method is defined, then forming a program number by using the value `0x0011` for the high-order 16 bits, and the high-order 16 bits of the CRC for the low-order 16 bits of the program number. The version number is the low-order 16 bits of the CRC.

Successful replies are sent using the Courier `ReturnMessageBody` specified in section 4.3.3 of the Courier specification. The `procedureResults` field contains the return value, if any, followed by the `INOUT` and `OUT` parameter values, if any.

User exceptions are signalled using the `AbortMessageBody` specified in section 4.3.4 of the Courier specification. The `errorValue` field contains a value greater than 0, which is the ordinal value of the particular exception being signalled in the list of exceptions specified for this method. The `errorArguments` field contain the exception value, if any.

System exceptions (of exception type `ilu.ProtocolError`) are signalled using the `RejectMessageBody` message of section 4.3.2. The `rejectionDetail` field of the message contains the `ProtocolError` detail.

13.1.3.2 Mapping of Standard Types -- Courier Layer 2

The mapping of ILU types into Courier types is accomplished primarily by using the appropriate Courier Layer 2 representation for that type.

`Short integer` and `integer` types are represented with the Courier `Integer` and `Long Integer` types. `Long integer` types are represented as an `integer` followed by a `cardinal`.

`Short cardinal`, `byte`, and `cardinal` types are represented with the Courier `cardinal`, `cardinal`, and `long cardinal` types, respectively. `Long cardinal` types are represented as a big-endian (most significant 16 bits first) Courier array of 4 `cardinals`.

As the Courier protocol does not have any mapping for floating point values, `short real` numbers are passed as a Courier `long cardinal`, `real` numbers are encoded as a big-endian array of two Courier `long cardinal` values, and `long real` numbers are encoded as big-endian array of four Courier `long cardinal` values.

`Array` values are encoded as Courier one-dimensional `arrays`. If the array is multi-dimensional, it is encoded as a flat rendering into a single-dimensional array in row-major order (the last specified index varying most rapidly). If the array is of type `byte` or `short character`, the contents of the ILU value are packed into a Courier array of `unspecified` two values per array element, so that the Courier array is half the length of the actual ILU array.

`Record` values are encoded as Courier `record` values.

`Union` values of union types whose discriminant type can be mapped to a 16-bit value type in the range [0,2^16-1] are passed as Courier `choice` values. Other unions are passed as a Courier `long cardinal`, followed by the value of the union's indicated type (if any).

`Enumeration` values are encoded as Courier `enumeration` values.

`Boolean` values are encoded as Courier `boolean` values.

`Sequence` values are encoded as Courier `sequences`, except for several special cases. Sequences of *N* `short characters` or `bytes` are encoded as either a Courier `cardinal`, for sequences with limits less than 2^16, or `long cardinal`, for sequences with no limits or limits greater than 2^16-1, value of *N*, followed by `(`*N*`+1)/2` values of Courier `unspecified`, each such value containing two `short character` or `byte` values, packed in big-endian order.

`Optional` values are encoded as an Courier `boolean` value, followed by another encoded value, if the Boolean value is `TRUE`.

Instances of an `object` type are encoded as either zero (in the case of a method discriminant of a singleton type), or one values of ISL `short sequence of short character`. CORBA Nil object references are represented as a zero-length `short sequence of short character`.

13.1.4 The OMG Internet Inter-Orb Protocol (IIOP)

This section describes the mapping of the abstract ILU protocol into the specific on-the-wire protocol prescribed by the OMG's CORBA `Internet Inter-ORB Protocol` (`IIOP`), version 1.0.

A protocol info string for the `IIOP` version 1.0, with the ILU-to-IIOP mapping version 1, has the form `iiop_1_0_1`.

The `IIOP` is a concurrent protocol; it may be used on top of either a reliable, boundaried or non-boundaried, transport stack.

13.1.4.1 Message Mappings -- GIOP

ILU request and reply messages are mapped to GIOP Request and Reply messages fairly directly. The byte order used is that native to the machine on which the message is being formed. A zero-length service context is always sent.

In a Request message, the operation name is the ISL operation name for the method, with all hyphen characters in the operation name changed to underscore characters. The Principal field is always sent as a zero-length field.

The GIOP CancelRequest, LocateRequest, MessageError, and CloseConnection messages are never sent by ILU, though one or more of them may be used in the future. ILU will send GIOP LocateReply messages in response to LocateRequest messages.

13.1.4.2 Mapping of Standard Types -- GIOP

The mapping of ILU types into IIOP types is accomplished primarily by using the mapping for the corresponding CORBA type.

`Short integer` and `integer` types are marshaled as CORBA `short` and `long` types. `Long integer` types are represented as an `integer` followed by a `cardinal`.

`Short cardinal`, `byte`, and `cardinal` types are marshaled as the CORBA `unsigned short`, `octet`, and `unsigned long` types, respectively. `Long cardinal` types are marshalled as two CORBA `unsigned long` values, and the byte order of the message determines which is marshalled first.

`short real` numbers are passed as CORBA `float` values. `real` numbers are passed as CORBA `double` values, and `long real` numbers are encoded as big-endian array of 16 bytes.

`Array` values are encoded as CORBA `array` values.

`Record` values are encoded as CORBA `struct` values.

`Union` values are encoded as CORBA `union` values.

`Enumeration` values are encoded as CORBA `enum` values.

`Boolean` values are encoded as CORBA `boolean` values.

`Sequence` values are encoded as CORBA `sequence` values.

`Optional` values are encoded as a CORBA `sequence` of the base type, with an upper limit of one value.

`Object` values are passed as an `IIOP` Interoperable Object Reference (IOR), containing at least an Internet Profile. The IOR may also contain an ILU Profile. In the case of the Internet Profile, the object key contains four strings, separated by NUL (zero octet) characters. The first string is always `"ilu"`. The second string is the most specific type ID of the object (in case some intervening ORB decides to re-write the IOR's type_id field). The third string is the server ID of the object's server. The fourth string is the instance handle of the object.

13.1.5 The Hyper Text Transfer Protocol (`HTTP`)

`HTTP` in ILU allows an ILU application to interact with an existing Web resource. That is, Web Browser to ILU, ILU to Web Server, and general ILU to ILU over `HTTP` is possible. `HTTP` is a non-concurrent protocol.

For `HTTP` interaction with existing web services, an ILU application must be able to not only get an object (a surrogate actually) representing the resource. It must also have some means by which to specify the `HTTP` headers and entity body that should be sent with the request. Similarly, an ILU server functioning as a `HTTP` accessible Web resource must be able to set status, header and entity body content.

Arbitrary programmers interpretations of these `HTTP` components cannot be generally mapped into `HTTP`. A specific signature is needed for the `GET HEAD` and `POST` methods so that the ILU implementation of the `HTTP` protocol can know how to map arguments into actual `HTTP` format. In addition, a way is needed to distinguish these methods intended for use with existing Web services from other methods that may happen to have the same name but different signatures.

This need is addressed by defining a specific type of object that has declarations for how an application should structure the arguments / return values for the `GET HEAD` and `POST` operations. Any `GET HEAD` or `POST` operation invoked on an object that is an instance of this base type (or an instance of a type derived directly or indirectly from that base type) has a particular signature that the ILU protocol implementation knows how to map to `HTTP`. This type, called `Resource` in the `http` interface, is defined in the http.isl file, and any application wishing to supply Web compatible objects should make the objects a direct or derived instance of it. A server for objects acessible via `HTTP` should be created with the protocol info string http_1_0, and should use the tcp transport.

A version of `HTTP` 1.0 that supports connection persistence (i.e. Connection: Keep-Alive headers) can be used by specifying the protocol info string http_1_0p. In this case, Connection: Keep-Alive headers will be automatically appropriately generated on both the client and server sides. Client side connections obtained from an http URL, e.g. from 'Object_of_SBH' fuctions, will use persistence automatically (assuming the server they're connecting to supports it of course). Client side connections obtained from ILU style SBH's will use whatever protocol the server advertizes.

See the `httest` example for a sample use of `HTTP` in ILU.

A method named `GET HEAD` or `POST`, invoked on an object that is a direct or indirect instance of the `Resource` type, automatically has its Request and Response mapped to/from `HTTP` in a manner compatible with existing Web services. The fairly straightforward mapping from the ILU `http` Interface to `HTTP` Protocol is outlined described below:

```
ILU Method Name                        Method name in Request's Request line
```

`(if using a Proxy server, scheme + location of object +) ILU Object ID + any params/queries present in the Request.URI field`	`Request-URI in Request's Request line`
`Request.headers`	`Headers in Request`
`Request.body`	`Entity-Body in Request`
`Response.status`	`Status-Code and Reason-Phrase in Response's Status-Line`
`Response.headers`	`Headers in Response`
`Response.body`	`Entity-Body in Response`

The HTTP implementation will automatically insert a Content-Length header when necessary and possible, and takes care of the colon separators between header names and values. It will also deal with older servers that sometimes omits the CR from the required CRLF line termination.

Note that existing Web tools (e.g. browsers) will always send the 'path' of the resource. On the ILU HTTP end, this means that the object identifier will always begin with a forward slash. For example, asking a browser to retrieve http://www.foo.bar.com/hello.html where www.foo.bar.com is serviced by an ILU HTTP server, will result in that server trying to invoke a GET operation on the object whose object identifier is /hello.html. Omitting any path info, i.e. asking the browser to retrieve http://www.foo.bar.com would result in a GET on an object whose object identifier is simply /.

Regarding the Request-URI field on the client side, it is really only necessary to put in any 'param's and/or 'queries'. Any path information in this field is just ignored, since the path info needed is to form the request is based on the object's instance handle. So for example, a client my simply put the string ";param1;param2?query" into the Request-URI field instead of "http://www.foo.bar.com/hello.html;param1;param2?query".

If operations need to occur through a proxy server, the environment variable ILU_HTTP_PROXY_INFO should be set to the proxy server name, colon, and port number e.g. ourproxyserver.foo.bar.com:8000.

For other situations, i.e. general ILU to ILU communication that just happens to be occurring over HTTP, the mapping is still consistent with HTTP protocol, but a more general format is used. ILU specific information such as the ilu_Server ID is placed in a header, and the marshaling of arguments is done entirely within the entity body. In keeping with some idea of human readability, marshaled arguments, with the exception of potentially huge byte-vectors, are encoded as readable ASCII strings - e.g. 3.1416 encodes as "3.1416". Readers concerned about utmost efficiency should note that for general ILU-ILU communication, another protocol such as ONC RPC is a much better choice than the current HTTP implementation. The HTTP protocol implementation could however be easily changed to use a more efficient encoding, similar to what's used in ONC RPC for example. The following grammer describes the on-the-wire mapping of ILU's HTTP when it is used for general ILU to ILU communication.

```
( ----------------------------------------------------------------
  ILU HTTP on the wire encoding for general ILU  method invocation
  (i.e. when object is not of iluhttp.Resource type)
  ---------------------------------------------------------------- )


( ------------------------------------------------------------
  basic items
  ------------------------------------------------------------
)

OCTET               = <any 8-bit sequence of data>
CHAR                = <any US-ASCII character (octets 0 - 127)>
ALPHA               = <any US-ASCII uppercase letter "A".."Z" or "a".."z">
DIGIT               = <any US-ASCII digit "0".."9">
SP                  = <US-ASCII SP, space (32)>
CRLF                = <US-ASCII CR, carriage return (13) followed by LF,
linefeed (10)>
HYPHEN              = <US-ASCII hyphen (45)>
HEX                 = "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c"
| "d" | "e" | "f" | DIGIT
ISO-LATIN-1-NO-NULL = <any ISO-LATIN-1  character other than NUL>

hexbyte         = HEX HEX
cstring         = ISO-LATIN-1-NO-NULL [cstring]
digits          = DIGIT [digits]  ( interpreted as base 10 )
octets          = OCTET [octets]
nameString      = ALPHA [ nameStringChars ]
nameStringChars = nameStringChar [nameStringChars]
nameStringChar  = ALPHA | DIGIT | HYPHEN


( ------------------------------------------------------------
  Typed Values
  ------------------------------------------------------------
)

sbh = (string binding handle - follows IETF rules for URLs)

typedValues    = typedValue [typedValues]

typedValue     = optionalValue | value

optionalValue  = optional [value]

value          = primitiveValue | constructedValue
```

```
( ------------ Primitive values --------------------- )

primitiveValue = integer   | shortInteger   | longInteger  |
                 cardinal  | shortCardinal  | longCardinal |
                 byte      | boolean                       |
                 real      | shortReal      | longReal     |
                 character | shortCharacter | pickle

integer        = [-] digits CRLF

shortInteger   = [-] digits CRLF

longInteger    = integer integer (high word followed by low word)

cardinal       = digits CRLF

shortCardinal  = digits CRLF

shortCardinals = shortCardinal [shortCardinals]

longCardinal   = cardinal cardinal (high word followed by low word)

byte           = 0x hexbyte CRLF

boolean        = "TRUE" CRLF |
                 "FALSE" CRLF

real           = <as printed by  "%.64g format in printf> CRLF

shortReal      = <as printed by  "%.64g format in printf> CRLF

longReal       = 16 CRLF (number of bytes in a long Real)
                 hexbyte hexbyte hexbyte hexbyte hexbyte hexbyte hexbyte
hexbyte
                 hexbyte hexbyte hexbyte hexbyte hexbyte hexbyte hexbyte
hexbyte

character      = shortCardinal

shortCharacter = byte

pickle         = digits CRLF (number of bytes in the pickled value)
                 octets CRLF (pickled contents)


( ------------ Constructed values --------------------- )

constructedValue = array    | sequence    | record | union |
                   optional | enumeration | object
```

```
array     = typedValues (Nothing is sent specially to indicate an array,
                         just the array's contents)

sequence  = digits CRLF (the sequence length, sequence contents follow)
            typedValues

record    = typedValues (Nothing is sent specially to indicate a record,
                         just the record's contents follow)

union     = digits       CRLF (union discriminator)
    typedValue

optional  = "OPT_PRESENT"      CRLF  |
            "OPT_NOT_PRESENT"  CRLF

enum      = digits CRLF (the enumeration value)

object    = nilObject | nonNilObject

nilObject = "7" CRLF "NIL_OBJ" CRLF

nonNilObject = digits  CRLF (number of characters in the objects sbh)
       sbh     CRLF (the objects sbh)


( ------------------------------------------------------------
  requests
  ------------------------------------------------------------
)

methodName  = nameString

requestBody = arguments

arguments   = typedValues

objectDiscriminant = cstring (discriminant object's instance handle)

serverId = cstring (ID of discriminant object's server)

request     = methodName SP objectDiscriminant SP "HTTP/" majorVersion "."
minorVersion CRLF
      "ILU_ServerID:" SP serverId CRLF
              "Content-Length:" SP digits CRLF    (number of bytes in the
requestBody)
              CRLF
              [requestBody]
```

```
( -------------------------------------------------------------
  responses
  -------------------------------------------------------------
)

responseBody = returnvalues

returnvalues = typedValues

response     = normalResponse | exceptionResponse



normalResponse     =  "HTTP/" majorVersion "." minorVersion SP "200" SP "OK"
CRLF
                      "Content-Length:" SP digits CRLF    (number of bytes
in the response Body)
                      CRLF
                      [responseBody]


exceptionResponse = protocolExcpeption   |  ( problem with protocol itself
)
                    userException           ( user method raised exception
)


exceptionIndex         = digits (which exception of the method an excep-
tion is)

internalExceptionIndex = digits (index of internal ilu_ProtocolException)


userException =  "HTTP/" majorVersion "." minorVersion SP exceptionIndex
SP "Non_Protocol_Exception" CRLF
                      "Content-Length:" SP digits CRLF    (number of bytes
in the typedValue response Body)
                      CRLF
                      [typedValue]   (the exception's contents)


protocolException =  "HTTP/" majorVersion "." minorVersion SP internalEx-
ceptionIndex SP "Protocol_Exception" CRLF
                      "Content-Length:" SP "0" CRLF    (number of bytes in
the response Body)
                      CRLF
```

```
( -------------------------------------------------------------
  End of File
  -------------------------------------------------------------
)
```

13.1.6 The World Wide Web HTTP-NG Protocol (w3ng)

This section describes the mapping of the abstract ILU protocol into the specific experimental on-the-wire protocol specified by the World Wide Web's HTTP-NG project, `w3ng`. See `http://www.w3.org/Protocols/HTTP-NG/Group/PDG/wire-protocol/WD-HTTP-NG-wire.html` fora full description of the protocol. The protocol is currently under design, so changes to it should be anticipated.

A protocol info string for the `w3ng` version 1.0, has the form `w3ng_1.0`.

The `w3ng` protocol is a concurrent protocol; it must be used on top of either a reliable, boundaried transport stack.

13.1.7 The Default Protocol and Transport

Server applications can create ports without specifying which protocol and transport to use. In this case, *the default protocol and transport* will be used. Which protocol and transport are the defaults is a consequence of which are included in the configuration step performed at the start of installing ILU. The first protocol from the following list that is included in the configuration is the default: sunrpc, courier, iiop, http. The default transport is chosen to be a configured transport appropriate for the default protocol. That is, sunrpcrm over TCP for sunrpc or courier, bare TCP for iiop and http.

13.2 Transports

A transport stack consists of a sequence of transport layers. The last, or ''bottom'', layer does some kind of low-level I/O; the other layers are ''filters'' or ''modifiers'' on the transport services provided by the lower part of the stack. That is, every tail of a transport stack implements an abstraction called simply ''a transport'' (in English; in C, it is `ilu_Transport`); each transport layer (except the bottom) implements a transport in terms of another transport (the one implemented by the rest of the stack).

Transport stacks are used in two roles, active and passive. A client uses a transport stack in the active role to contact a server; a server uses a transport stack in the passive role to wait for connection requests from clients. The passive role for a transport stack is called a *mooring*.

A transport stack is specified by a sequence of strings, each one of which specifies a transport layer. This section is a catalog of the built-in kinds of transport layers and their specification strings.

Some transports convey delimited messages (where each message is a byte sequence), others simply convey a byte sequence (that must be parsed into messages by something else). The former are called boundaried, the latter are not. Some RPC protocols require a boundaried transport, others require non-boundaried transports.

Some transports are reliable, and some aren't. Unreliable transports are deprecated in ILU, but included for interoperability with existing software that uses only unreliable transports. With these transport, messages may be delivered more than once. The ILU implementation of UDP on the server side filters out multiple receipts of the same request. Asynchronous methods may not be called over this transport mechanism, as reliable delivery of the request packet cannot be recognized by the client side. Non-asynchronous methods use the reply message as an acknowledgement that the request was received.[4]

13.2.1 TCP

A `TCP` transport layer is reliable, not boundaried, and goes on the bottom.

A `TCP` transport is specified by a *transport-info* of the form `tcp`*_host_port*`[`*_buffersize*`]`.

The *host* needs to convey an IP address. The *host* can either be a dotted decimal notation of an IP address (e.g., `13.2.116.14`), or be a hostname that can be mapped into an IP address. The optional *buffersize* parameter is an unsigned integer; it can be used to control the size of buffers allocated for use in the transport, where buffer usage is allowed by the higher-level transport or protocol.

In ILU, a TCP mooring always bound to a particular IP address. However, the *host* in a `TCP` *transport-info* used to create a mooring can use special notations to mean "pick any IP address of this host". Those notations are `0`, `0.0.0.0`, and `localhost`. When one of these notations is used, the ILU runtime picks an IP address and binds the mooring to that address. These special notations are replaced with either a name or address of the host when an SBH is produced; an SBH cannot contain such special notations. `0` and `0.0.0.0` are replaced by an address; `localhost` is replaced by a name, if possible, and perhaps an address otherwise. The first replacement name considered is "the hostname" of the machine. Exactly what this is (and how it is set) is system specific, but you should beware that it may or may not be a Fully Qualified Domain Name. If it can be converted to an IP address by the usual means, and a socket bound to that address, that name is used. Otherwise, the replacement is "127.0.0.1" (the loopback address --- an address that means "this machine" everywhere) or "localhost" (the canonical name for the loopback address) and the mooring's socket is bound to 0.0.0.0 (the "any" address). We bind to 0.0.0.0 rather than 127.0.0.1 because some systems (e.g., Linux) won't let us bind to 127.0.0.1.

---

[4] Query: can requests be larger than the UDP packet size? [No] How then are they segmented? [They're not; replies aren't either] Note: This should probably be replaced by a reliable UDP protocol, in which each message is acknowledged by the receiver. This would allow use of asynchronous methods over UDP. Of course, ONC RPC would not cooperate.

If you use a host name instead of an address, think about how widely it can be interpreted. If it's a Fully Qualified Domain Name, the client has to be able to use the DNS to resolve it --- but not all systems include DNS support. If it's not a FQDN, the resulting SBH can only be distributed within the organization that manages the name's mapping.

The *port* can either be a decimal string identifying a specific port (which, of course, must not be used for anything else), or, for a mooring, be `0`, which constitutes a request for a new, unused port. In the latter case, a decimal string for that port will be substituted in the transport-info.

Examples are:

```
tcp_augustus_0
tcp_13.2.116.14_12321
tcp_localhost_12321
tcp_0_0
```

13.2.2 UDP

A `UDP` transport is boundaried, not reliable, and on the bottom.

A `UDP` transport supports only messages that are no longer than the maximum size of a `UDP` packet; an attempt to send a longer message will cause an I/O error to be raised in the sender.

One could imagine creating a reliable boundaried transport layer, either using `RDP` (an `IP` protocol similar to `UDP`; see RFC 1151) or building on `UDP`. Of course, such a transport layer could not be used to communicate with ONC RPC/UDP peers.

A `UDP` transport is specified by a *transport-info* of the form `udp_`*host*`_`*port*. The *host* and *port* convey IP host address and port just as for `TCP`.

Examples are:

```
udp_augustus_0
udp_13.2.116.14_12321
udp_localhost_12321
udp_0_0
```

13.2.3 SunRPC Record Marking

A `SunRPC Record Marking` transport is reliable and boundaried, and is built on top of some other transport that is reliable and not boundaried. A `SunRPC Record Marking` transport layer is specified by a *transport-info* of the form `sunrpcrm`.

The canonical form is:

```
sunrpcrm
```

13.2.4 World Wide Web Consortium MUX Transport

The `World Wide Web Consortium MUX` transport filter is reliable and boundaried, and will run only on top of the TCP transport. It supports multiple virtual transport channels over a single TCP/IP transport. Each channel can be used with a different server and/or protocol and/or transport stack. Both outgoing and incoming channels can be used with the same TCP/IP connection, so the MUX protocol supports callbacks through a firewall. The MUX protocol also supports message fragmentation and interleaving, and provides application-level flow control. The protocol supported is as documented in `http://www.w3.org/Protocols/MUX/WD-mux-961023.html`,but with the modifications suggested in `http://lists.w3.org/Archives/Member/w3c-mux/msg00039.html`.A `World Wide Web Consortium MUX` transport layer is specified by a *transport-info* of the form `w3mux_`*CHANNEL-ID*, where *CHANNEL-ID* specifies an 18-bit channel number. If the *CHANNEL-ID* is specified as `0` when creating a mooring, a random available channel ID will be assigned.

The canonical form is:

```
w3mux_0
w3mux_258049
```

13.2.5 Generic Security Service (GSS)

`GSS` transport layers may be added to a transport stack to provide some form of authenticated connection. It uses the IETF Common Authentication Technology Working Group's Generic Security Service (GSS) API to add various flavors of security to the messages that flow back and forth over the transport. Generally speaking, each outgoing message will be ''wrapped'' by the standard GSS routine `gss_wrap`, and each incoming message will be ''unwrapped'' by the standard GSS routine `gss_unwrap`. This transport also includes a mechanism for identifying callers that is integrated with the specific security scheme being used.

Use of this transport requires linking against a GSS library, implemented according to the ANSI C mapping for the GSS spec, and against an implementation of the specific GSS scheme being used.

The `security` transport layer is reliable and unboundaried, and requires a reliable, boundaried, transport stack below it. It is specified, on the server side, by a string of the form `gss_1_`*SCHEME-NAME*, where *SCHEME-NAME* identifies some specific GSS security scheme. Scheme names are typically dotted-decimal strings, representing OIDs for specific schemes. Two special names are also understood, "Xerox.ILU.GSS.NIL" and "Xerox.ILU.GSS.SSL". Examples are:

`gss_1_Xerox.ILU.GSS.NIL` -- use security with the ILU GSS NIL scheme
`gss_1_1.2.840.113550.9.1.3` -- another way of saying the same thing

14 Security

There are a number of components that make up a successful security policy for an application based on a distributed object system. Only a few of them deal with the distributed object system. This document gives an overview of some of the issues, and explains how they may be achieved with the mechanisms provided in ILU 2.0. For a fuller discussion of the issues, see the OMG Security Working Group's white paper on security, `http://www.omg.org/docs/1994/94-04-16.ps`. There are a number of things which are often lumped under the umbrella of 'security'. They include:

- *identities* -- some partitioning and naming scheme that lets the system separate one principal from another;
- *authentication* -- a way of proving that an entity claiming some identity actually has the right to that identity;
- *communications authentication* -- some way of proving that a message supposedly sent by some identity actually was sent by that identity;
- *communications integrity* -- some way of knowing that a received message is the same message that was sent, that no bits were altered along the way;
- *communications privacy* -- some way of concealing the contents of a message from all but the intended recipient;
- *authorization* -- a mechanism for deciding whether an identity should be allowed access to some data or function;
- *auditing* -- a way of remembering, securely, who did what to what.

Some of these items are more complex than they appear to be at first glance. For example, the notion of authentication includes not only the mechanisms needed to check identities, but also the more social problems of effective password and key management. Authorization and auditing may include various schemes of payment and chargeback, for access to services. They are also inter-related; communications privacy is a kind of authorization problem.

ILU provides two major hooks for an application to use in implementing a security policy. The first is a flexible notion of identities. The second is an interface to communications security which allows authentication, integrity, and privacy to be provided in all message traffic between two ILU-connected programs. We believe that with these two mechanisms, arbitrary security policies can be implemented with ILU.

14.1 Identities

Identities are ways in which authorization principals in a security system are named. They can be various; human names, social security numbers, computer login accounts, and credit cards are all examples of identities. ILU provides a few prespecified identity types, one of which (the GSS identity) is flexible and user-extensible. In addition, the `ilu_IdentityType` data type in the kernel exports a meta-object protocol for defining new identity types, which can be used by various application-specific protocols and transports.

ILU also defines a *passport*, which is a data structure which can contain an arbitrary number of identities. Identities are communicated during calls through an ILU interface by having the caller pass a passport as part of the call, so that the implementor of the true method can find out which principal is making the call. This passport transfer can be accomplished automatically as part of the wire-protocol used in the call, or on a per-connection basis by the transport machinery, or explicitly as a parameter of the call.

The identity types provided in ILU 2.0 are:

- `ilu_ConnectionIdentity` -- a string identifying the connection over which the call was made;
- `ilu_SunRPCAuthUnixIdentity` -- a data structure (see '`runtime/kernel/iluxport.h`' for details) identifying the (unauthenticated) user ID, group ID, and UNIX group memberships on the (putative) calling machine (this identity type is only available if ILU has been configured with support for the `sunrpc` protocol);
- `ilu_GSSIdentity` -- an opaque value which identifies the caller via one or more GSS security mechanisms (this identity type is only available if support for the `security` transport layer has been configured into ILU).

Passports, and the identities in them, are represented in different ways in different language-specific runtimes. This section contains only a brief description of the various identity types; refer to the ILU documentation for your programming language to see how passports and identities are manipulated in that language.

The 'connection' identity can be used to display the apparent IP host address of the caller; due to the possibility of IP packet spoofing in IPv4, that address should not be relied upon for security purposes. Another problem with using connection identities in server applications is that ILU will close them and re-constitute them at need, which means that a client might in the extreme case use a different connection for every call. This renders the connection identity somewhat inappropriate for identifying particular client-server 'sessions'.

The 'Sun RPC Unix Authentication' identity follows the 'system authentication' scheme described in Internet RFC 1831. It contains the host IP address, the UNIX user ID, and the UNIX group ID of the caller, and a list of the UNIX groups to which the caller belongs. Note that the user ID and group ID are relative to the caller's host, not the server's host. This information is only passed if the caller is running on a UNIX machine, and if the protocol used in the connection is the `sunrpc` protocol. Note that this information may be easily spoofed if the communications channel is insecure.

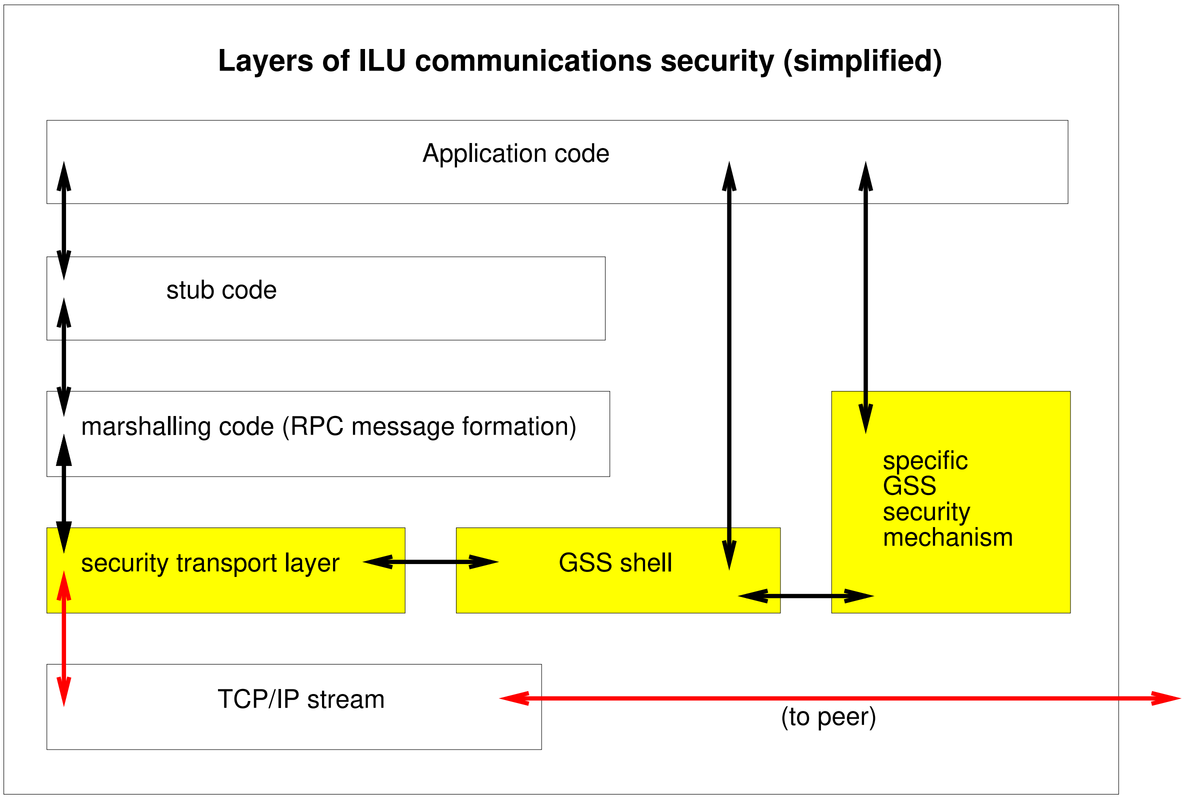
The 'GSS' identity may contain multiple different identities, in the form of GSS 'names'. GSS naming is flexible and extensible, so should be able to represent any type of name desired.[1] These identities may be reliable, even in the face of insecure networks, if they have been negotiated and communicated with an appropriately secure GSS security mechanism. Any effective security mechanism for use with ILU should use some form of GSS identity. ILU provides ways to retrieve parameters of the GSS identity, including the `gss_name_t` of the caller, the lifetime of the security context, and the security mechanism that has validated the identity and protected the communications. There are also routines to convert a `gss_name_t` value to a printable string, and to create a GSS identity given a value of type `gss_cred_id_t`. Further information on the GSS can be found in the ILU distribution, in the directory '*ILUSRC*`/GSS/doc/`'.

14.2 Communications Security

ILU provides for transparent communication between ILU address spaces when an operation is invoked on a foreign (supported outside the calling address space) object reference. There is a generic scheme for securing inter-address-space communication, using the IETF Common Authentication Technology working group's Generic Security Service (GSS) API. (The ILU distribution also provides an implementation of the GSS, which contains a meta-object protocol allowing specific security mechanisms to be registered with and used by the GSS shell implementation.) Inside the generic mechanism, specific security mechanisms can be selected to give various degrees of protection.

The GSS-based inter-address-space security provides message integrity and/or message privacy services, depending on which specific services are requested of the specific security mechanism selected for use. All parts of the message are protected; all messages on a secured transport are sent as record-marked GSS tokens. Multiple tokens may be sent for a single RPC-level message; some security mechanisms may insert empty padding messages to further confuse traffic analysis attacks. This inter-address-space security may also be used for ILU 'transport-buffer-based' object persistence mechanisms, but it is not clear to us what security implications this has.

[1] The current ILU GSS implementation only allows one name per GSS identity. This restriction will be lifted in the future.



ILU communications security is orthogonal to the choice of RPC protocol or transport mechanism. The ILU transport stack consists of the following pieces: an application layer, which calls into a caller-side stub layer, which calls into RPC-mechanism-specific marshalling code, which calls into one or more transport filters, arranged in a stack. For example, an application in C++ may call into the stubs for some specific interface, which call down into the RPC message marshalling code, which calls down to the compression transport layer (passing it either whole RPC messages or fragments thereof), which calls down to the security transport layer (which turns byte streams passed from above into security data tokens), which calls down to a simple record-marking layer (which puts 32-bit length headers on each GSS data token), which calls down to the TCP/IP transport layer (which actually writes the bytes to the TCP/IP stream). Thus, each message passed between address spaces in this example is a record-marked GSS token containing an encrypted signed compressed fragment of a full RPC message. The fragment may be a 'improper' fragment -- a whole RPC message. Note that this is only one possible ILU transport stack; the layers may be arranged in other orders, within some constraints. In particular, our current design requires the security transport layer to be immediately above a GSS token record-marking layer, which in turn must be immediately above the actual transport layer (the layer which actually writes to the TCP/IP buffers, for instance). The GSS token record-marking layer may adapt to the particular security scheme being used; in particular, if the wire format of the security mechanism already specifies record-marking information, the record-marking layer will simply interpret that information, rather than adding any new information to the packets passing through it.

14.2.0.1 Kernel Servers

ILU objects are managed by *kernel servers*. There may be multiple kernel servers per address space. Each kernel server 'owns' some set of ILU object instances, and manages certain functionality for those instances. One item of functionality managed by an kernel server is communication with kernel servers in other address spaces. This inter-kernel-server communication is always performed between a 'true' kernel server, or *TKS*, and its counterpart in another address space, the 'surrogate' kernel server, or *SKS*. Instances in a TKS are those which actually contain implementations for the operations of the instances' types, so the TKS may be thought of as the 'server-side' kernel server. A SKS manages surrogate or proxy instances for some subset of the instances managed by its corresponding TKS, so the SKS may be thought of as the 'client-side' kernel server. There may be many SKSs corresponding to any single TKS, but there may only be one SKS for each TKS in any one address space. All inter-address-space communication is between a SKS and its corresponding TKS.

14.2.0.2 Creating Ports

Selection of security mechanisms for access to instance operations is done by calls to the ILU kernel on the server side which create 'ports' on true kernel servers. Each port describes a particular communication mechanism which SKS's may use in communicating with the TKS. Each port may independently select an RPC protocol and transport stack, and (if the transport stack includes the security transport layer) identify which security mechanism to use, and which server-side identity to use in establishing security contexts via that security mechanism. Communication with that TKS by any of its corresponding SKS's may then be accomplished through any of the ports on the TKS. In the call to create a port, the server-side application code passes a passport containing an ILU GSS identity, which is used as the identity of the principal offering the service, and put into the connection information in the string binding handle of objects on that server.

14.2.0.3 Client Connections to Ports

When a client decides to use a particular object, it consults the string binding handle of the object to discover communication parameters for that object, including the parameters of the various ports of the TKS for the object. The client then creates a SKS for the TKS, if one does not already exist in its address space, and indicates which types of communication channels it wishes to use. A communication channel is an abstraction which incorporates the notions of RPC protocol (say, the OMG `IIOP`, though others are possible with ILU), message transport (say, TCP/IP), and possible message transformations such as compression or security. Note that TKS's can force a particular communication channel type by either only opening one port, or only telling this particular client about one of its ports.

When a client opens a communications channel of a particular type, the SKS initialization code automatically creates the proper RPC protocol object and transport layer stack on the client side, initializing each layer in the transport stack as appropriate. If the transport stack includes the security transport layer, it also initializes this layer. The security transport layer is written against the GSS C API, using no extensions. The initialization code retrieves the client's identity (from the passport of the client), the server's identity (one of the parameters of the port, and passed as a parameter to the secure transport layer's initialization code), and the specific security mechanism to use (specified as a string, which may be either the GSS-specified string form of the security mechanism OID, or an ILU nickname for it (an optional feature)). The init code then calls `gss_init_sec_context()` with the client credentials, the server identity, and the security mechanism. Any context token output by that call is passed through the lower-level transport layers to the TKS, where it is passed to `gss_accept_sec_context()`. Any return token is passed back, etc, until the security context is established. This sequence of messages is called the security context handshake.

Note that at this time, and in all future uses of the channel, the actual transport mechanism, TCP/IP, is used only to convey GSS tokens back and forth, typically with some simple record-marking protocol which may either be provided by the particular wire format of the security mechanism (if any), or via an additional optional record marking layer between the security layer and the TCP/IP layer. No recognizable RPC messages are ever sent on the TCP/IP connection.

14.2.0.4 Calls

After the client has opened a channel to the TKS, calls may be made on objects in that TKS. When a call is made, the client calls down into the marshalling stubs with an indication of which channel is to be used for the call. If no port is specified, the SKS makes the choice according to simple defaulting rules. Typically, a SKS has only one channel open to its TKS, and the client accepts it by default. The marshalling stubs build up the appropriate RPC message, and dispatch it, either whole or in fragments, to the top layer of the transport stack. For the purpose of this discussion we'll use the term fragment, but it is important to remember that a whole message is also a valid 'fragment'. Each layer of the transport stack transforms the fragment in some way, and passes it to the next lower layer. In particular, the security transport layer takes the fragment and applies `gss_wrap()` to it, using the security context that was negotiated when the channel was set up. The wrap procedure may perform either message integrity on the fragment, or message privacy, or both. This produces a GSS data token, which is passed to the next lowest layer for possible additional transformation and eventual transport to the TKS. On the true side, the stack is reversed; a GSS data token coming in is passed to `gss_unwrap()`, which either verifies the message integrity check, or decrypts the token to produce a fragment, or both, and sends it up to the next highest level.

14.3 Using the GSS Transport Filter

To use GSS security in an ILU application, either client or server, your ILU installation configuration must have included the GSS transport filter. You will also need an implementation of the GSS, with the C API specified in the now obsolete internet draft '*ILUSRC*`/GSS/doc/draft-ietf-cat-gssv2-cbind-01.txt`'. The ILU distribution comes with such an implementation, in the directory '*ILUSRC*`/GSS/`', and will build and use this implementation unless another is explicitly specified at configuration time.

To export an object via a secure connection, the true module will need to (1) establish an identity via some GSS scheme, and (2) create a true kernel server with a port that uses the GSS transport filter in its `tinfo` stack. In this discussion, we'll use the ANSI C mappings and funtions.

To establish an identity, the true module must first acquire GSS credentials for some identity + security-mechanism pair; see the documentation on the GSS in '*ILUSRC*`/GSS/doc/`' to see how to do this, or take a look at the code in '*ILUSRC*`/examples/test1/srvr.c`'. Then the module will create an ILU *GSS identity*, by calling the C API function `ILU_C_AcquireGSSIdentity`. This identity must be placed in an ILU passport, by calling either of the functions `ILU_C_CreatePassport` or `ILU_C_AddIdentity`. The server module then creates a TKS by calling `ILU_C_InitializeServer`, passing in as parameters a transport info list which includes the GSS transport filter, and the passport. A suitable transport info list should contain a record-marking layer immediately below the GSS transport filter, and may need another record-marking layer above the GSS transport filter, depending on the RPC protocol being used. For example, if the protocol being used is `sunrpc`, an appropriate transport info list would be `{ "sunrpcrm", "gss_1_Xerox.ILU.GSS.NIL", "sunrpcrm", "tcp_0_0" }`. For the CORBA `IIOP`, which does not need a record-marking layer below it, a suitable layering would be `{ "gss_1_Xerox.ILU.GSS.NIL", "sunrpcrm", "tcp_0_0" }`.

You will note that there are two parameters in the transport info string for the GSS transport filter, `"1"` and `"Xerox.ILU.GSS.NIL"`. The first is the ILU version number of the GSS transport filter; the second identifies the security mechanism being used 'behind' the GSS shell. This security mechanism identifier is typically specified as a dotted-decimal representation for the ISO OID for that security mechanism. For example, to specify the Kerberos 5 security mechanism, you would use the string `"1.3.5.1.5.2"`, which is the assigned OID for the Kerberos 5 security mechanism. The GSS transport filter can recognize a small number of nicknames for security mechanisms (currently just `"Xerox.ILU.GSS.NIL"` and `"Xerox.ILU.GSS.SSL"`). You should consult your GSS implementation to see what specific security mechanisms are supported, and what the ISO OIDs for them are.

When the client makes a call on an object exported via a secure channel, the client may have to provide an identity of its own, if the security mechanism selected by the true module requires one. To do this, the client performs the same steps the true module went through: calling the GSS to acquire credentials, and then calling the ILU kernel to create a passport containing a GSS identity. The client passes the passport in a language specific way, when making the call. For the ILU C bindings, the client should call `ILU_C_SetPassportContext`, before making the call. If the object has been exported via a secure port, the first call on the object will result in the client address space negotiating a secure channel to the server address space, using the identities passed in by the client, and provided at port creation time by the server. Subsequent calls on that object will use the same security context and identities.

In the true method code for an object exported via a secure channel, the GSS identity of the caller will be available in the `callerPassport` field of the `ILU_C_ENVIRONMENT` parameter for the call; in C, use the function `ILU_C_CallerIdentity` call to retrieve the caller's passport, then use the `ILU_C_FindIdentity` function to obtain a GSS identity from the passport, and the `ILU_C_DecodeGSSIdentity` function or the `ILU_C_DisplayIdentity` function to display it. (Note that the `ILU_C_GSSNameToString` function is provided as a convenience.) The true method can use this identity to do authorization, accounting, and other application-specific security functions. If the security mechanism doesn't require an identity from the caller, the special GSS identity of `anonymous` (`"2.16.840.1.113687.1.2.1:<* anonymous *>"`) will be passed as the GSS identity for the caller.

14.3.1 GSS Namespace Schemes

The GSS system allows many different namespace schemes to co-exist, though specific security mechanisms may require the use of specific namespace schemes. As of release 2.0alpha13, the following namespace schemes are supported in ILU:

- `ANONYMOUS` -- required by the GSS specification, this namespace scheme allows only a single name, for the anonymous identity. The OID for this scheme is 2.16.840.1.113687.1.2.1.
- `RFC822` -- allows names specified in simplified RFC 822 notation: `account@host`. The OID for this namespace scheme is 1.2.840.113550.9.1.4. The ILU GSS tranport filter also supports an alias for this namespace, `Xerox.ILU.GSS.RFC822`.
- `X509` -- Uses X.509 identifiers as names. The OID for this scheme is 1.2.840.113550.9.1.2. The ILU GSS transport filter also supports an alias for this namespace, `Xerox.ILU.GSS.X509`. *This mechanism is not available in the freeware version of ILU, due to U. S. export control considerations. Xerox users of ILU may apply to* `ilu-core@parc.xerox.com` *for a copy.*

14.3.2 GSS Security Schemes

The GSS system allows many different security schemes to co-exist. As of release 2.0alpha13, the following security schemes are supported in ILU:

- `NIL` -- the NIL security scheme. Though it exercises most parts of the GSS implementation, this scheme is not authenticated via strong cryptography, so provides no real security. However, it does successfully transmit RFC 822 names through a negotiated context. The OID for this scheme is 1.2.840.113550.9.1.3. The ILU GSS transport filter also supports an alias for this namespace, `Xerox.ILU.GSS.NIL`.
- `SSL` -- negotiates a security context using X.509 names and SSL-based security. Uses strong cryptography to secure context. Requires RSAREF and libdes. The OID for this scheme is 1.2.840.113550.9.1.1. The ILU GSS transport filter also supports an alias for this namespace, `Xerox.ILU.GSS.SSL`. *This security mechanism is not available in the freeware version of ILU, due to U. S. export control considerations. Xerox users of ILU may apply to* `ilu-core@parc.xerox.com` *for a copy.*

15 ILU Customization

15.1 Introduction

ILU includes a number of internal interfaces that allow various functionality of the ILU kernel library to be replaced by user functionality.

- In a single-threaded application, the entire ‘‘main loop’’ can be replaced; in a threaded application, features of the thread system can be described to the ILU kernel so that the kernel will use them.
- New messaging protocols and inter-address-space transport mechanisms can be registered for use by an application.
- Object incarnation routines can be registered for ‘kernel servers’, so that true instances can be created on reference, rather than before reference.
- URL parsing routines can be registered for various URL schemes, so that application-specific object URLs can be used.
- New identity schemes can be defined and registered with the kernel to support application-specific authorization and accounting schemes.
- Handler routines can be registered to deal with `malloc()` failures in application-specific ways.
- Handler routines can be registered to perform application-specific actions when errors are signalled.
- If debugging is enabled at configuration time, specific sets of debugging messages can be selected at run time, and an application routine can be defined to intercept and handle debugging messages.

Most of these interfaces are defined in two ILU source files, ‘*ILUSRC*`/runtime/kernel/iluxport.h`’ and ‘*ILUSRC*`/runtime/kernel/iluntrnl.h`’. We will not attempt to duplicate that documentation here, to avoid the inevitable errors when documentation is provided in two different forms; rather, this section of the manual will provide sketches of the interfaces, and refer the reader to the appropriate header files. For any discrepancies noted between the material here, and the material in the header files, the header files should be assumed to the ‘truth’.

15.2 Event Loops and Threads

Every ILU address space uses either real threads, or some sort of event dispatching loop to simulate threads. ILU is thread-safe internally, and by default will continue to check its usage for even when event dispatching is used. Since many different thread systems and event dispatching loops exist, ILU provides interfaces to allow the user to describe the particular one that they’re using to the ILU kernel. See section Chapter 12 [Threads and Event Loops], page 191 for more information on these topics. See ‘*ILUSRC*`/runtime/kernel/iluxport.h`’ for documentation of the interfaces.

15.3 RPC Protocols and Data Transport Mechanisms

The ILU remote procedure call mechanism operates in layers.

- The application, at the highest layer, calls down into language-specific, interface-specific stub code, passing language-specific, method-specific values;
- that stub code calls down into an interface-independent language-specific runtime layer, passing method-independent, but language-specific values;
- that LSR calls down into language-independent, interface-independent ILU 'kernel' code, passing simple C values;
- the kernel calls into the code for a specific RPC protocol, such as ONC RPC, passing C values;
- the protocol code calls down into transport-mechanism code, passing 'messages' as opaque byte sequences;
- the transport code, which may consist of several layers, may perform various transformations on the messages, and finally conveys them to the peer transport layers.

ILU includes registration mechanisms to allow applications to add to the kinds of RPC protocols and data transport mechanisms that can be used.

15.3.1 RPC Protocols

Each `ilu_Protocol` object reflects a particular mapping of the abstract ILU RPC protocol onto a specific externally-defined RPC protocol. (See Section 13.2.5 [Protocols and Transports], page 216 for a discussion of the abstract ILU protocol, and how it is mapped to the ONC RPC protocol, and to the XNS Courier protocol.)

New RPC message protocols can be added to the ILU kernel by writing a new `ilu_Protocol` object, and calling the ILU kernel function `ilu_RegisterProtocol().` register it. The structure and requirements of an `ilu_Protocol` object are defined in the file '*ILUSRC*`/runtime/kernel/iluprotocol.h`'; the methods of the protocol are considered to be 'inside' the ILU kernel, and must therefore conform to all ILU locking and error conventions. The locking conventions are discussed in '*ILUSRC*`/runtime/kernel/iluxport.h`'; the error conventions are documented in '*ILUSRC*`/runtime/kernel/iluerror.h`', and pre-defined errors are documented in '*ILUSRC*`/runtime/kernel/iluerrs.h`'.

Various examples of ILU protocols are available for study:

- the ONC RPC mapping ('*ILUSRC*`/runtime/kernel/sunrpc.c`'
- the XNS Courier protocol ('*ILUSRC*`/runtime/kernel/courier.c`')
- the OMG Internet Inter-Orb Protocol ('*ILUSRC*`/runtime/kernel/iiop.c`')
- the World Wide Web Hyper-Text Transport Protocol ('*ILUSRC*`/runtime/kernel/http.c`')

15.3.2 Transport Filters

In general, ILU protocols form 'messages' consisting of sequences of bytes, which are then passed to the ILU transport layer to be conveyed to another address space. The transport layer itself is composed of one or more ILU *transport filters*, each of which handles the message in turn. These filters are either communication filters, such as the filters which actually convey messages via TCP/IP or UDP/IP, or transformation filters, which alter the message and pass it to another transport filter, such as the ONC RPC record-marking filter, or the secure transport filter.

Each transport filter is either *reliable* or *unreliable*. All transformation filters are reliable; communication filters may or may not be reliable. A communication filter is reliable if it guarantees that any messages handed to it for transport will be reliably delivered to the other end of the communication connection. This in turn means that the communication mechanism used by the transport will take care of timeouts, retries, etc., internally, so that the ILU application need not worry about these itself. Unreliable communication filters are those which may require ILU participation in timeout and resending of messages to achieve reliable delivery.

Each filter is also either *boundaried* or *non-boundaried*. Boundaried filters are those which can comprehend and preserve message boundaries. Non-boundaried filters simply deal in chunks of bytes and have no way to recognize or preserve message boundaries. Various protocols and filters may have requirements as to whether the next filter below it in the communication stack is boundaried or non-boundaried.

New transport filters may be registered with the ILU kernel by calling the kernel function `ilu_RegisterTransport()`, described in '*ILU*`/runtime/kernel/iluxport.h`', with the name of a new transport filter and the address of a routine which returns an instance of the new transport object type. Implementing a new transport object type actually consists of implementing several related object types, including `ilu_TransportCreator`, `ilu_TransportClass`, and `ilu_Mooring`. These object types are defined in the file '*ILUSRC*`/runtime/kernel/ilutransport.h`'.

As with protocols, the methods of the transport filter are considered to be 'inside' the ILU kernel, and must therefore conform to all ILU locking and error conventions. The locking conventions are discussed in '*ILUSRC*`/runtime/kernel/iluxport.h`'; the error conventions are documented in '*ILUSRC*`/runtime/kernel/iluerror.h`', and pre-defined errors are documented in '*ILUSRC*`/runtime/kernel/iluerrs.h`'.

Examples of transformation filters may be found in '*ILUSRC*`/runtime/kernel/sunrpcrm.c`', which is a boundaried filter implementing ONC RPC's TCP/IP record marking scheme, and '*ILUSRC*`/runtime/kernel/securit` which is a non-boundaried filter implementing message integrity, sender authentication, and message privacy. Examples of communication filters may be found in '*ILUSRC*`/runtime/kernel/newtcp.c`', which is a non-boundaried reliable filter implementing data communication via TCP/IP, '*ILUSRC*`/runtime/kernel/udp.c`', which is a non-boundaried unreliable filter implementing data communication via UDP/IP, and '*ILUSRC*`/runtime/kernel/inmem.c`', which is a boundaried reliable filter implementing intra-address-space communication via memory buffers.

15.4 Object Incarnation Procedures

ILU true objects live in *kernel servers*, a kernel data structure that handles communication and other aspects of the object implementation. When an object reference is received from another address space, the kernel server is responsible for mapping this reference to an actual object. Normally, the kernel server simply consults an internal hash table for an object corresponding to a specified 'instance handle'; however, an application may register an application-specific callback function to be used instead. This allows on-the-fly creation of objects, which is often vital when handling many objects. Actual in-memory representations of the objects can be garbage-collected, then dynamically re-incarnated when needed by a client.

The application registers this functionality by creating an implementation of an `ilu_ObjectTable` object, and passing that implementation as a parameter to `ilu_CreateTrueServer` when creating the kernel server. Typically, `ilu_CreateTrueServer` is called directly only by a language-specific runtime; the actual application would work with object tables via whatever mechanism is exported by the language-specific runtime. Check the documentation for your particular language runtime for more information.

15.5 Object URLs

ILU regards string binding handles generically as a way of encoding four pieces of information: the instance handle for an object, the server ID for an object, the most-specific type ID (MSTID) for an object, and communication information about how to communicate with that object, which we call *contact-info*. It further restricts them to conform to the URL syntax specified in the World Wide Web Consortium and IETF standard RFC 1738 (`http://www.w3.org/pub/WWW/Addressing/rfc1738.txt`). But thisstill allows ILU to support any number of URL schemes, which we define as some way of encoding these four pieces of information which conforms to the URL syntax rules.

The default URL scheme is called `ilu:`, and encodes the information as

`ilu:`<*server-id*>`/`<*instance-handle*>`;`<*MSTID*>`;`<*contact-info*>

Most of these elements consist of US-ASCII strings with various additional constraints. The strings are encoded in what is called the *SBH element encoded form*: the set of alphanumeric characters plus the 4 characters DOLLAR ('`$`'), HYPHEN ('`-`'), PERIOD ('`.`'), and PLUS ('`+`') are represented by the character itself; other characters are escaped via the mechanism specified in RFC 1738: each is represented with 3 characters, a PERCENT ('`%`') character followed by two hexadecimal digits giving the US-ASCII character code for the escaped character.

The non-encoded form of the <*server-id*> and <*instance-handle*> strings may contain any character except for US-ASCII NUL.

The non-encoded form of the <*MSTID*> consists of the following

<*type-id-scheme*>`:`<*type-id*>

where the <*type-id-scheme*> consists of US-ASCII alphanumeric characters, and any constraints on <*type-id*> are specified by the <*type-id-scheme*>.

The <*contact-info*> is not encoded in the same way as the other fields. Rather, it consists of a series of communication info fields, separated by SEMICOLON ('`;`') characters. Each communications info field has the form

<*protocol-info*>`@`<*transport-layer*>`[=`<*transport-layer*>`...]`

where each of the <*protocol-info*> and <*transport-layer*> elements contain SBH element-encoded strings. The non-encoded form of these strings has an additional constraint: each must begin with the name or identifier for the protocol or transport layer it specifies, optionally followed by an UNDERSCORE ('_') character and any parameters for the protocol or transport. The name of the protocol or transport may not contain any UNDERSCORE ('_') characters. There are no additional ILU constraints on the formats used to represent parameters for the protocol or transport.

An application can register a parser for one or more application-specific URL schemes by calling the function `ilu_RegisterSBHParser`. It takes as an argument a function which will accept a URL string, and return the four components required by ILU. For instance, you might want to use a URL scheme for the OMG CORBA `IIOP` something like

`iiop_1_0://`<*hostname*>`:`<*port*>`/`<*server-id*>`/`<*ih*>

which can be considered to contain an instance handle of <*ih*>, a server ID of <*server-id*>, an implicit object type of `IDL:omg.org/CORBA/Object:1.0`, and contact-info of `iiop_1_0_1@tcp_`<*hostname*>`_`<*port*>.

Or, you might want to use an `HTTP` URL for an ILU object which is exported via the `HTTP` ILU protocol. Suppose that the normal ILU string binding handle for the object was

```
ilu:tcp_1.2.3.4_20000//http_obj0;ilu:Ilu_Http_Type;http_1_0@tcp_1.2.3.4_20000
```

An alternate form which would be compatible with Web browsers would be

```
http://1.2.3.4:20000//http_obj0
```

with an implicit server ID of `tcp_1.2.3.4_20000`, an implicit MSTID of `ilu:Ilu_Http_Type`, an instance handle of `/http_obj0`, and contact-info of `http_1_0@tcp_1.2.3.4_20000`.

See the '*ILUSRC*`/runtime/kernel/iluxport.h`' for details on how to use `ilu_RegisterSBHParser`.

15.6 Identity Types

As discussed in Section 14.1 [Identities], page 217, application-specific identity types can be registered with the ILU kernel for use with various authorization and accounting schemes, and to support various forms of security in wire protocols and transports. An application does this by creating a new value of type `ilu_IdentityType`, as specified in '*ILUSRC*`/runtime/kernel/iluxport.h`', and calling the kernel function ilu_RegisterIdentityType. After this is done, values of the new identity types may be used. The major use for these identity types is to work together with new application-specific RPC protocols and message transports (described above), to implement various security and access policies for distributed systems.

Note that the mere act of registering a new identity type with the ILU kernel will not cause values of that identity type to be automatically transmitted in ILU calls. This will only happen if an appropriately designed transport or protocol, which knows to do this, is also used. For experimental purposes, we provide a switch will will cause the various flavors of Sun RPC implemented for ILU to automatically pass *one* specific identity type. To enable this, set the environment variable ILU_SUNRPC_PREFERRED_IDENTITY to the name of the identity type to be passed automatically before running any of your ILU programs.

By default, the Sun RPC protocols will automatically pass the UNIX identification of the caller (the user id, group id, host IP address, and list of groups to which the caller belongs). The identity type for this information is called "SunRPCAuthUnixIdentity". To prevent its being passed automatically, set the environment variable ILU_NO_SUNRPC_UNIX_AUTH to any value before running your ILU programs.

15.7 Malloc Failure Recovery

ILU uses a number of internal interfaces to allocate and free memory, such as ilu_malloc(), ilu_free(), and ilu_realloc(). These functions wrap calls to the standard malloc(), etc., in wrappers that allow for better error handling. They are documented in

'*ILUSRC*/runtime/kernel/iluxport.h'.

Applications can register callback functions to handle malloc failures, in two ways. The kernel function ilu_AddFreer() allows registration of routines which can be called to free up memory, to allow a malloc call to succeed. The kernel functions ilu_SetMemFaultAction() and ilu_SetMemFaultConsumer() allow applications to determine what action should be taken if a malloc failure occurs.

15.8 Error Reporting

ILU includes a comprehensive error-signalling system in the kernel library, which is documented in '*ILUSRC*/runtime/kernel/iluerror.h'. In addition, the kernel library contains many calls to _ilu_Assert, which check that various kernel invariants are maintained. When a runtime assertion fails, the kernel may either call an application-specified failure handler, set by a call to ilu_SetAssertionFailureConsumer(), or take one of three default actions, chosen by a call on ilu_SetAssertionFailureAction(). The three default actions are (1) to generate an illegal instruction trap, and thus coredump; (2) to exit with some error code; and (3) to enter an endless loop, calling sleep() repeatedly. The third action is the default action; the intent is to stop the program with all invalid data intact on the stack, and network connections intact, so that a debugger may attach to the 'live' process.

15.9 Debugging Interfaces

The ILU kernel contains a large number of debugging print statements, which document various things going on inside the kernel. The specific things printed may be controlled by calling either `ilu_SetDebugLevel()` or `ilu_SetDebugLevelViaString()`. The specific bits which can be specified to `ilu_SetDebugLevel()`, or names which can be specified to `ilu_SetDebugLevelViaString()`, are documented '*ILUSRC*`/runtime/kernel/iludebug.h`'.

All debugging messages are displayed via calls to the kernel function `ilu_DebugPrintf()`. Normally, this routine simply calls `vfprintf (stderr, ...)` to actually output the messages. However, this can be changed to call some application-specific message output system by calling `ilu_SetDebugMessageHandler()`, documented in '*ILUSRC*`/runtime/kernel/iluxport.h`'. Two special values are defined for, and accepted by, `ilu_SetDebugMessageHandler()`; the value `ILU_DEFAULT_DEBUG_MESSAGE_HANDLER` causes the debug system to revert to the original output handler; the value `ILU_NIL_DEBUG_MESSAGE_HANDLER` causes the debug system to simply discard any debugging messages.

Debugging output can be directed to a file, by calling `ilu_SendDebugOutputToFile()` with a filename as an argument. The file will be created, and debugging messages will be written to it.

16 Debugging ILU Programs

This document describes some of the common errors that occur with the use of ILU, and some techniques for dealing with them.

16.1 C++ static instance initialization

Our support for C++ currently depends on having the constructors for all static instances run before `main()` is called. If your compiler or interpreter doesn't support that, you will experience odd behavior. The C++ language does not strictly mandate that this initialization will be performed, but most compilers seem to arrange things that way. We'd like to see how many compilers *do not*; if your's doesn't, please send a note to `ilu-bugs@parc.xerox.com` telling us what the compiler is.

ILU uses the static-object-with-constructor trick to effect per-compilation-unit startup code. In certain cases you'll want to ensure that a certain compilation unit's initialization is run before another's. While C++ defines no standard way to do this, most compilers work like this: compilation units are initialized (static object construtors run) in the order in which they are given to the link-editor. We (`ilu-bugs@parc.xerox.com`) want to hear about any exceptions to this rule.

16.2 ILU trace debugging

Note for Windows users: Please refer to the chapter "Using ILU with Microsoft Windows" to see how ILU trace debugging is handled for Windows applications.

ILU contains a number of trace statements that allow you to observe the progress of certain operations within the ILU kernel. To enable these, you can set the environment variable `ILU_DEBUG` with the command `setenv ILU_DEBUG "`*xxx*`:`*yyy*`:`*zzz*`:..."` where *xxx*, *yyy*, and *zzz* are the names of various trace classes. The classes are (as of December 1997) `packet`, `connection`, `incoming`, `export`, `authentication`, `object`, `sunrpc`, `courier`, `dcerpc`, `call`, `tcp`, `udp`, `xnsspp`, `gc`, `lock`, `server`, `malloc`, `mainloop`, `iiop`, `http`, `error`, `sunrpcrm`, `inmem`, `security thread`, `lsr`, `type` and `binding`. The special class `ALL` will enable all trace statements: `setenv ILU_DEBUG ALL`. The special class `MOST` will enable all trace statements except `lock`, and `malloc`: `setenv ILU_DEBUG MOST`. The environment variable `ILU_DEBUG_FILE` may be used to direct debugging output to a file. The function `ilu_SetDebugLevelViaString(char *trace_classes)` may also be called from an application program or debugger, to enable tracing. The argument *trace_classes* should be formatted as described above.

`ILU_DEBUG` may also be set to an unsigned integer value, where each bit set in the binary version of the number corresponds to one of the above trace classes. For a list of the various bit values, see the file '*ILUHOME*`/include/iludebug.h`'. Again, you can also enable the tracing from a program or from a debugger, by calling the routine `ilu_SetDebugLevel(unsigned long trace_bits)` with an unsigned integer argument.

The routine `ilu_SetDebugMessageHandler` allows an application to specify an alternate routine to be called when an error or debugging message is to be printed.

`void` **ilu_SetDebugMessageHandler** (`void` (`*`*handler*) (`char *formatSpec, va_list args`)) [ILU kernel]

Locking: unconstrained

Registers *handler* with the ILU kernel to be called whenever a debugging or error message is output via `ilu_DebugPrintf`, instead of the default handler, which simply prints the message to `stderr`, using `vfprintf`. Two special constant values for *handler* are defined, `ILU_DEFAULT_DEBUG_MESSAGE_HANDLER`, which will cause the default behavior to be resumed, and `ILU_NIL_DEBUG_MESSAGE_HANDLER`, which will cause debugging and error messages to be simply, silently, discarded.

16.3 Debugging ISL

16.3.1 Use of `islscan`

The `islscan` program is supplied as part of the ILU release. It runs the ISL parser against a file containing an interface, and prints a "report" on the interface to standard output. It can therefor be used to check the syntax of an interface before running any language stubbers.

16.3.2 The ISLDEBUG environment variable

Setting the environment variable `ISLDEBUG` to any value (say, `"t"`), before running any ILU stubber or the program `islscan`, will cause ILU's parser to print out its state transitions as it parses the ISL file. If you're having a serious problem finding a bug in your ISL file, this might help.

16.4 Bug Reporting and Comments

Report bugs (nah! -- couldn't be!) to the Internet address `ilu-bugs.parc@xerox.com`, or to the XNS address `ILU-bugs:PARC:Xerox`. Bug reports are more helpful with some information about the activity. General comments and suggestions can be sent to either `ILU@parc.xerox.com` or `ILU-bugs`.

Often the our first reply to a bug report is a request for a typescript that shows the bug occurring, with all trace debugging turned on. If that doesn't make it clear to us, our second reply may be a request for a stack trace, with printouts of relevant variables and data strutures. Including these things in your bug report may speed the cycle of interactions.

16.5 Use of `gdb`

When using ILU with C++ or C or even Common Lisp, running under the GNU debugger `gdb` can be helpful for finding segmentation violations and other system errors.

ILU provides a debugging trace feature which can be set from `gdb` with the following command:

```
(gdb) p ilu_SetDebugLevel(0xXXX)
ilu_SetDebugLevel:  setting debug mask from 0x0 to 0xXXX
$1 = void
(gdb)
```

The value *XXX* is an unsigned integer as discussed in section 3. The debugger `dbx` should also work.

We are in the midst of installing a consistent new way of handling rutime failures into the ILU runtime kernel. This new way involves the kernel reporting the failure to its caller; the old way involves combinations of panicking, reporting to the user (not the caller) via a printed message, and fragmentary reporting to the caller. Every time a runtime failure is noted the new way, the procedure `_ilu_NoteRaise` in '*ILUSRC*`/runtime/kernel/error.c`' is called; this procedure thus makes a good place to set a breakpoint when debugging. Most runtime failures occur due to genuine problems; some occur during normal processing (e.g., end-of-file detection).

16.6 Error handling

Ideally, the ILU runtime would report all failures to the application, in the way most appropriate for the application's programming language. Sadly, this is not yet the case.

The ILU runtime kernel has three kinds of runtime failures:

1. memory allocation failures from which the kernel cannot proceed;
2. internal consistency check failures, from which the kernel cannot proceed; and
3. internal consistency check failures, which the kernel is prepared to report to the ILU language-specific runtime veneer (which, hopefully, would in turn report the failure to the applicaiton).

The second kind is being eliminated. The first kind is being reduced, and might also be eliminated.

The application can specify how each of these three kinds of runtime failures is to be handled. The choices are:

1. Print an explanatory message and then explicitly trigger a `SEGV` signal by attempting to write to protected memory. This is useful for generating core dumps for later study of the error.
2. Print an explanatory message and then exit the program with an application-specified exit code.
3. Print an explanatory message and then enter an endless loop, which calls `sleep(3)` repeatedly. This option is useful for keeping the process alive but dormant, so that a debugger can attach to it and examine its ''live'' state. This is the default action for all three kinds of failures.
4. Invoke an application-supplied procedure (without printing anything first).
5. Report the failure out of the kernel, without printing anything first (this option is available only for the third kind of failure).

An application can change the action taken on memory failures by calling `ilu_SetMemFailureAction` or `ilu_SetMemFailureConsumer`.

`void` **ilu_SetMemFailureAction** (`int` *mfa*) [ILU kernel]

Locking: unconstrained

Calling this tells the ILU kernel which drastic action is to be performed when `ilu_must_malloc` fails. -2 means to print an explanatory message on stderr and then coredump; -1 means to print an explanatory message on stderr and then loop forever in repeated calls to `sleep(3)`; positive numbers mean to print an explanatory message on stderr and then `exit(`*mfa*`)`. The default is -1.

`typedef void (*) (const char *file, int line)` [ILU kernel]
ilu_FailureConsumer

A procedure that is called when the ILU kernel can't proceed. This procedure must not return.

`void` **ilu_SetMemFailureConsumer** (`ilu_FailureConsumer` *mfc*) [ILU kernel]

Locking: unconstrained

An alternative to `ilu_SetMemFailureAction`: this causes *mfc* to be called when `ilu_must_malloc fails`.

Similarly, an application specifies how unrecoverable runtime consistency check failures are to be handled by calling `ilu_SetAssertionFailureAction` or `ilu_SetAssertionFailConsumer`, which are exactly analogous to the procedures for memory failure handling. For recoverable consistency check failures, an application can call `ilu_SetCheckFailureAction` or `ilu_SetCheckFailureConsumer`.

`void` **ilu_SetCheckFailureAction** (`int` *cfa*) [ILU kernel]

Locking: unconstrained

Calling this tells the runtime which action is to be performed when an internal consistency check fails. -3 means to raise an error from the kernel (without necessarily printing anything); -2 means to print an explanatory message to stderr and then coredump; -1 means to print and then loop forever; non-negative numbers mean to print and then `exit(`*cfa*`)`; others number reserved. The default is -1.

`typedef void (*) (const char *file, int line)` [ILU kernel]
ilu_CheckFailureConsumer

A procedure for handling an internal consistency check failure. If this procedure returns, the consistency check failure will be raised as an error from the kernel.

`void` **ilu_SetCheckFailureConsumer** (`ilu_CheckFailureConsumer` *cfc*) [ILU kernel]

Locking: unconstrained

An alternative to `ilu_SetCheckFailureAction`: this causes *cfc* to be called (and no printing); if *cfc* returns, an error will be raised from the kernel.

16.6.1 Decoding reportable consistency check failures

For language mappings consistent with CORBA, the third kind of failure is reported as an occurrence of the CORBA system exception `internal`, with a minor code that encodes the filename and line number where the consistency check occurs. The coding is this: 10,000*hash(filename, 32771) + linenum + 1,000. The directory part, if any, is stripped from the filename before hashing. To aid in decoding these minor codes, ILU includes the program `decoderr`, which is used like this:

```
% decoderr 269211234
269211234 = line 234, file $ILUSRC/runtime/kernel/call.c
```

If a reportable consistency check failure occurs in a file not anticipated in the construction of `decoderr`, you'll see something like this:

```
% decoderr 60612345
60612345 = line 1345 in unknown file (that hashes to 6061)
```

The program `iluhashm` can be used to hash given filenames, so you can search a set of candidates for the mysterious hash code:

```
% iluhashm 32771 ../cpp/foobar.cpp ../cpp/barfoo.cpp
/* Generated at Mon Dec 11 22:44:47 1995
   with modulus 32771 */
{      6061, "../cpp/foobar.cpp"},
{     13273, "../cpp/barfoo.cpp"},
```

16.7 Common Problems - Questions

Users often run into the same difficulties other users have had. This section lists some of these common problems, and describes the possible cures.

Problem: A server cannot publish an object or a client cannot lookup an object.

Discussion: When using the shared file approach for simple binding, the machines on which the client and server programs run must have some shared filesystem. Each program must also have the environment variable ILU_BINDING_DIRECTORY set to a directory within that file system where the publications will be written and read.

Problem: It seems that ILU is contacting the wrong server, or if I look at the SBH's for objects that I know are coming from one source, ILU thinks they're from someplace else.

Discussion: This is usually caused by creating multiple ilu_Server's (e.g. in C, the thing you get back from ILU_C_InitializeServer (...)) that have the same server ID. The server ID should be unique. To understand why, consider what ILU does when an non-local object reference (an SBH) comes in off the wire. ILU looks at the reference and checks to see if it has a surrogate ilu_Server with that name. If not, ILU creates one, and (important point) stores away the contact information for that ilu_Server. If it already has one with that name, ILU assumes that that is the ilu_Server for the object - it doesn't check to see if the contact info is different. Thus, operations directed at objects who are served by that particular ilu_Server will always be directed at the ilu_Server that ILU saw first. [ILU could potentially keep track of multiple contact infos, but that still wouldn't help to disambiguate where operations should be directed.] This is why in many of the example programs, you see server ID's being created using some combination of a fixed string, and a the name of the host the process is running. Of course, if you run multiple instances of the example on the same machine, you would want to also incorporate some process or thread information. You can also simple let ILU generate a server ID for you, that is unique with a high degree of probability.

Problem: My process 'A' has an object reference to an object 'O' in process 'B'. Process 'B' exits, and then restarts. Even though the server name and object identifier for 'O' are the same as the first time around, process A is unable to perform operations on 'O'.

Discussion: The answer here is similar to the answer to the "It seems that ILU is contacting the wrong server... " problem. You're probably letting ILU choose the server's port number by either letting ILU use it's defaults, or by specifying a 0 in the port field in the transport specification when creating the ilu_Server. ILU in process 'A' caches the contact information from the first process 'B'. When process 'B' comes back up, the port number is different. You can specify what port should be used in the transport information to prevent this from happening. For example, to always come up at port 1234, use "tcp_0_1234".

Problem: How come a lot of things seem broken in the C++ support!

Discussion: Yes, we know! The C++ support is in the process of being completely redone for ILU release 2.0. As such, we allowed the current support to remain in disrepair, and don't bother to fix much of anything in it. Use the current C++ support at your own risk!

Problem: I'm having problems importing ILU into Python.

Discussion: (Where ILUHOME represents where you installed ILU) You need to have the ILUHOME/lib directory on your PYTHONPATH environment variable. Also, ensure that ILUHOME/bin is also on your PATH environment variable.

Problem: I'm in Windows, and trying to build some of the examples and I get complaints that it doesn't know how to make some of the files.

Discussion: The Windows make files are not set up to run the language stubbers. You must run the stubbers manually before doing the make. e.g. c-stubber Test1.isl

Problem: I'm on Unix (most probably Digital's), and my program sometimes exits unexpectedly.

Discussion: You may be running into a problem where a PIPE signal is generated and the established action is to exit the program. In the ILU source file runtime/kernel/bsdutils.c, the function _ilu_HandleSigPIPE tries to set up the process to ignore SIGPIPE. However, it only does this if the initial SIGPIPE is not SIG_DFL. (You should see an error message if _ilu_HandleSigPIPE can't setup to ignore SIGPIPE) On some systems it has been noticed that even though the application did not explicitly set up a SIGPIPE handler, the initial SIGPIPE is not SIG_DFL, and the handler that runs terminates the program. A workaround to this problem is to either set the SIGPIPE handler to SIG_DFL yourself before the _ilu_HandleSigPIPE function runs, or set it to a handler that does nothing with the signal.

17 Installation of ILU

This document describes the installation of version 2.0alpha14 of the Inter-Language Unification (ILU) system.

If you succeed in installing ILU on a particular platform, we'd appreciate it if you could send a note to `ilu-core@parc.xerox.com` *telling us (1) what operating system you succeeded with, and what version of that OS, (2) which versions of what compilers you used, and (3) which version of ILU you used. We're accumulating a list of operating systems and compilers that work with ILU. If you had to make any changes to make it work on your system, please send them along, and we'll incorporate them into the next release.*

17.1 Installing on a Windows NT or 95 System

For information on Windows systems, see the "Using ILU with Microsoft Windows" section of the manual.

17.2 Installing on a UNIX System

17.2.1 Prerequisites

You will need an ANSI C compiler to build and install ILU, along with an ANSI C-compliant '`libc.a`'. Note that GNU `gcc` doesn't always work as an ANSI C compiler. The simple test we use to qualify a compiler is whether it can compile and link the following program without warnings or errors:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
int main(int ac, char **av) {
 int i = INT_MAX;
 char *p = (char *) malloc(1048);
 memmove(p, *av, strlen(*av)+1);
 printf("%s %d\n", p, i);
 return 0;
}
```

ILU requires the `imake` program from the MIT X Consortium release of the X Window System, version 4 or later. This is available via FTP from the ftp servers `ftp.x.org` on the East Coast, or `gatekeeper.dec.com` on the West Coast. You can also get it from `ftp://ftp.parc.xerox.com/pub/ilu/imake/imake.tar.gz`.

ILU normally provides support for a number of languages, currently ANSI C, C++, Java, Python, and Common Lisp (Franz Allegro 4.3), but the 2.0alpha release only contains solid support for ANSI C, Java, and Python. C++ support (sans nested modules and any/pickles) that complies with the CORBA 2.0 spec is available for Visual C++ users - it has received very limited testing . The old, flakey minimal C++

support is still provided. There is good support for Common Lisp in 2.0alpha, but it lacks some of the features provided for C, Java, and Python. There is rough support for Guile Scheme, contributed by Siemens Corporate Research, Inc. ILU support for the Perl programming language is available from Owen Taylor; see `http://www.msc.cornell.edu/~otaylor/ilu/` fordetails.

- If you wish to build the support for ANSI C, you will need a C compiler, and an ANSI C-compliant libc. But you already need that to build ILU.
- This release contains the old C++ support in '*ILUSRC*`/{stubbers,runtime}/cpp/`'. The old version of ILU C++ support has a number of problems. There are known leaks in the generated stubs, and in the runtime itself. We do not recommend serious use of the old C++ support, but it is included for use in testing other parts of the system. If you wish to build the support for C++, you will need a C++ compiler that conforms to at least version 2.0 of the C++ specification. ILU's old C++ does not use either C++ templates or exceptions. The GNU C/C++ compiler `g++` seems to work well with ILU. It has also been tested with Lucid's Energize `lcc` compiler, CenterLine's `CC` compiler, and Sun's `CC` compiler.
- If you wish to build the support for Common Lisp, you will need a copy of Franz Allegro Common Lisp, version 4.1 or later on Unix, or 3.0 or later on Windows.
- If you wish to build support for Python, you will need the Python 1.3 (or later) release, available via FTP from '`ftp://ftp.cwi.nl/pub/python/`'. When installing Python on your system, be sure to do a "make inclinstall", so that the include files are installed for ILU's use. It's also advisable to do "make libainstall", if you wish to try the 'multiple languages in the same address space' example.
- If you wish to build support for Java, our current release works against Sun JDK 1.1, Sun JDK 1.2, Microsoft Java SDK 2.0 and Microsoft Java SDK 3.1. See `http://java.sun.com/products/JDK/` for instructions on obtaining and installing the Sun JDK. If you are building for Linux, you should give the command line switch **--with-java-nopoll** to the configure script, to ensure the Java runtime uses the `select` system call instead of `poll` (which is present but decidecly inferior on Linux).ILU also supports various beta releases (some of which might require special handling). Support for applets or Netscape is not generally available.
- If you wish to build support for Guile Scheme, you will need to have both Guile 1.2 ('`ftp://prep.ai.mit.edu/pub/g 1.2.tar.gz`') and SLIB 2b1 '`http://www-swiss.ai.mit.edu/~jaffer/SLIB.html`'. Guile must be installed so that SLIB support works; try using `guile -c '(use-module '(ice-9 slib))'` to see if yours is configured properly.
- If you wish to build support for Perl, you will need Perl 5.004 or later. See `http://www.perl.com/` for details on acquiring and installing Perl.The Perl support is not provided directly in this ILU release; you have to download and unpack the Perl support tar file from `http://www.msc.cornell.edu/~otaylor/ilu/`.
- If you wish to build support for the zlib compression transport filter, you're out of luck in this release, as it's not provided. But when it is, you will need to get and install a copy of the zlib library, version 1.1.3, and specify its location to the configure script with the **--with-zlib=** command-line switch. See `http://www.cdrom.com/pub/infozip/zlib/` for details on acquiring and installing zlib.

- If you wish to build the SSL mechanism for GSS, you will need to first get and install copies of the RSAREF-2.0 and libdes 3.0 libraries, and specify their locations to the configure script with two command-line switches. For RSAREF, use **--with-rsaref=***LIB***:***INSTALLDIR* to specify the absolute filename of the library file (*LIB*) and installation directory (*INSTALLDIR*); the installation directory is the one that includes the `"source"` directory as a child. For libdes, use **--with-libdes=***LIB***:***INCLUDEDIR* to specify the location of the library file (*LIB*) and include directory (*INCLUDEDIR*); actually, it appears there is no directory structure in the libdes 3.0 distribution --- the include files are found in the source directory. Note that SSL support is not included in the freeware version of ILU.

17.2.2 Unpacking the Distribution

Begin by creating two directories: one, *ILUHOME*, to install the ILU in, and the other, *ILUSRC*, to unpack the sources in, and build the system in. It is often convenient if *ILUSRC* is a sub-directory of *ILUHOME*, but it is not necessary. At PARC, we use '`/import/ilu`' for *ILUHOME*, and '`/import/ilu/src`' for *ILUSRC*.

Copy the compressed tar file '`ilu-2.0alpha14.tar.Z`' or '`ilu-2.0alpha14.tar.gz`' to *ILUSRC*. Uncompress it if necessary with the `uncompress` or `gunzip` program:

```
% uncompress ilu-2.0alpha14.tar.Z
```

or

```
% gunzip ilu-2.0alpha14.tar.gz
```

Then unpack the tar file:

```
% tar xf ilu-2.0alpha14.tar
```

We suggest you then look at the ILU home page, `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`,to see whether a patch file for release 2.0alpha14 exists. If so, fetch that patch file and apply it according to the instructions at the top of the file. It's best to fetch the patch file using FTP instead of the Web; the additional line-ending transformations that Web browsers (particularly Netscape) apply can render the patches in the file worthless.

17.2.3 For the Impatient

You can try just unpacking it, and then typing

```
% make
```

at the top of the source tree. A minimal configuration using defaults will be built, sufficient for testing. This takes you up through step 2 of the `"Building"` section below.

17.2.4 Real Configuration

ILU uses the GNU autoconf system to configure the release, before building. The very simplest way to configure your system is type type

```
% ./configure
```

at the top of the source tree.

What will happen is that *ILUSRC*`/imake/configure` will go out and look along the value of your PATH environment variable for various executable programs. If it finds `cc`, it will assume that you want to build ANSI C support for ILU. If it finds `java`, it will assume that you want to build Java support for ILU. If it finds `cl` or `franz`, it will assume that you want to build Lisp support for ILU. If it finds `python`, it will assume that you want to build Python support for ILU. If it finds `CC` or `cxx` or `c++`, it will assume that you want to build C++ support for ILU. It will also assume that the first executable with an appropriate name is the one you wish to use for compiling programs in that language. By default, it will assume that you wish to include support only for using `ONC RPC` over `TCP/IP`. By default, it will assume that you do not wish to provide support for OMG IDL.

`imake` must be on your path, or in '`/usr/bin/X11/`', for the build to work properly.

17.2.4.1 Configuration Options

The program `configure` can be invoked with a number of command line options, to customize the build for your site. It actually supports more options than shown here, but these are the only options that will work at this point in the release process. For those options that begin **--enable-***FEATURE*, you can specify the switch either with **--enable-***FEATURE*, to enable the feature or option, or **--disable-***FEATURE*, to disable the feature or option.

- `--with-destdir=PATH` -- this should be the name of the directory you would like the files installed into, in the *`make Install`* step. The default is '`/usr/local/ilu`', which may be OK for your system.
- `--with-iluhome=PATH` -- this should be the name of the directory users will expect ILU to be installed under, the value of *ILUHOME*. The two options `--with-destdir` and `--with-iluhome` are provided separately because a directory may have two different names that are used to access it in different ways. At PARC, for instance, installation directories are often write-protected if named with their ordinary names, and a special name has to be given to enable writing in that directory. If your site does not have this type of restriction, the switches for `--with-destdir` and `--with-iluhome` should probably have the same values. Note that the default is '`/usr/local/ilu`', which may be OK for your system.
- `--prefix=PATH` -- if specified, this becomes the default for *ILUHOME* and *DESTDIR*, if they are not specified explicitly.

- `--with-binding-dir=PATH` -- a world-writable directory in a shared file system, for supporting the "simple binding" service described in Chapter 1. This option and the use of a binding service (see next item) are mutually exclusive.
- `--with-binding-server=REALM:HOST:PORT` -- an alternative to using the shared file system for simple binding, this says that a binding registration server for the binding realm called *REALM* will be available on port *PORT* of host *HOST*. The binding server will be built and installed as part of the build process, but still has to be started manually. This option and the use of a binding directory are mutually exclusive.
- `--enable-version-2-type-uids` -- compiles in support for documented type UID hash algorithm instead of the old default type UID hash algorithm. Versions of ILU configured with one algorithm are not generally compatible with versions configured with the other algorithm.
- `--enable-debugging` -- compiles in debugging printfs and a small amount of additional debugging code. Recommend enable. Enabled by default.
- `--enable-os-threads[=TYPE]` -- compiles in support for using either "solaris2", "posix", or "dce" (also known as POSIX.4) threads with the C support. If the TYPE isn't specified, it will attempt to guess the right default. The default is to enable thread support for platforms where the configure script knows thread support exists (currently only Solaris 2 (Solaris 2 threads), OSF 3 (DCE threads), OSF 4 (POSIX threads), AIX 4.x (POSIX threads), IRIX 6.x (POSIX threads), and linux-gnu (POSIX threads if '`/usr/lib/libpthread.a`' is present)). See the following section on ''Manual Fixups for Threading'', as well.
- `--enable-new-keywords-plain` -- normally, keywords added to ISL subsequent to 2.0alpha11 begin with "ILU", to avoid conflicts with user interfaces. However, by throwing this switch, this prefix will be omitted, and new keywords such as "string" and "state" will be recogized directly, rather than as "ilustring" and "ilustate".
- `--enable-type-support` -- includes support for runtime type registration. If either the `IIOP` or `w3ng` protocols is included, type support must be enabled. (Enabled by default).
- `--enable-pickle-support` -- includes support for a dynamically typed data type, called `pickle` in ILU and `any` in CORBA. (Enabled by default)
- `--enable-pickle-format-2` -- includes support for type 2 pickles. (The `pickle` type is supported with several formats, currently with version numbers 2 and 3.) (Enabled by default).
- `--enable-pickle-format-3` -- includes support for type 3 pickles. (The `pickle` type is supported with several formats, currently with version numbers 2 and 3.) (Enabled by default if the `w3ng` protocol is included; disabled by default otherwise.)
- `--enable-omg-idl-support` -- includes support for parsing interface description files written in OMG IDL. (Enabled by default.)
- `--enable-idl2isl-support` -- builds old ILU OMG IDL support program called `idl2isl`. You will need to have a C++ compiler (CFRONT 3.0 or later) on your path. `g++` seems to work fine. You will need to have fetched the source tar file from '`ftp://ftp.parc.xerox.com/pub/ilu/misc/idl2isl.tar.gz`',

and to have unpacked it in the *ILUSRC* directory, before configuring. The version of OMG IDL recognized by this parser is no longer current valid OMG IDL. (Disabled by default.)

- `--enable-xml-parser-library` -- includes a validating XML parser library. (Disabled by default.)
- `--enable-security` -- causes the GSS-based transport filter to be built, and also the GSS implementation. Default is disabled.
- `--enable-ssl-security` -- causes the SSL mechanism for GSS to be built. Not for public use (the public distribution of ILU does not contain the SSL mechanism in either source or binary form because of our inability to enforce export controls). Requires **--with-rsaref=...** and **--with-libdes=...**. Default is disabled.
- `--with-rsaref=`*LIB*`:`*INSTALLDIR* -- specifies the location of the RSAREF-2.0 library file and installation directory. The installation directory is the parent of the '`source`' directory. This is only needed if you are building the SSL mechanism for the GSS shell (the SSL mechanism is not distributed publicly due to our inability to implement export controls).
- `--with-libdes=`*LIB*`:`*INCLUDEDIR* -- specifies the location of the libdes 3.0 library file and include directory. This is only needed if you are building the SSL mechanism for the GSS shell (the SSL mechanism is not distributed publicly due to our inability to implement export controls).
- `--enable-sunrpc-protocol` -- compiles in support for using `ONC RPC` across address spaces. (Enabled by default.) This is probably our most flexible and most widely tested protocol.
- `--enable-courier-protocol` -- compiles in support for using `XNS Courier RPC` across address spaces. (Disabled by default) This protocol is more efficient than the `ONC RPC` protocol in terms of bytes-on-the-wire, but may have slightly higher marshalling and unmarshalling overhead.
- `--enable-corba-iiop` -- includes support for using the CORBA `Internet Inter-Orb Protocol` across address spaces. (Enabled by default) `IIOP` is the standard interoperability protocol for CORBA; people using ILU mainly for CORBA purposes should probably also specify `IIOP` as their default protocol (see **--with-default-protocol**).
- `--enable-http-protocol` -- compiles in support for using the World Wide Web `HTTP 1.x` between address spaces. (Disabled by default) This also provides support for implementing Web servers and clients with ILU.
- `--enable-w3ng-protocol` -- compiles in support for using a prototype of the HTTP-NG wire protocol between address spaces. (Disabled by default) This is a relatively new protocol which is probably the most efficient of all our supported protocols.
- `--with-default-protocol=`*PROTOCOL* -- specifies which 'wire protocol' to use as the default one for inter-process communication. The default default protocol is `"sunrpc"`. If you're interested in doing CORBA work with ILU, you may want to specify `"iiop"` as the default protocol, since IIOP is the standard CORBA interoperability protocol.
- `--enable-tcp-transport` -- includes support for using `TCP/IP` to transport messages. (Enabled by default.)

- `--enable-sunrpcrm-transport` -- compiles in support for the ONC RPC record-marking transport filter. This transport filter can be used with the `w3ng` or `sunrpc` protocols to provide record-marking on top of a byte-stream layer. (Enabled by default.)
- `--enable-w3mux-transport` -- compiles in support for using a prototype of the HTTP-NG multiplexing transport filter. (Disabled by default.) The current implementation of this transport requires the use of threads.
- `--enable-zlib-transport` -- If we shipped support for the zlib compression transport filter, this would compile in support for using `zlib` message compression on messages. But it's not currently in the release. If it were, you would also need to specify **--with-zlib=FOO** for this transport. (Disabled by default.)
- `--with-zlib=`*LIB*`:`*INCLUDE_DIR* -- specifies the location of the zlib library and include directory. This is only needed if you are including support for the zlib compression transport, which doesn't actually work yet, and is thus normally disabled.
- `--enable-c-support` -- includes ILU support for the ANSI C programming language. (Enabled by default.)
- `--enable-c-shared-library` -- causes shared-library version of the C runtime to be built. (Disabled by default.)
- `--enable-c-timing-statistics` -- if enabled, adds code to the C runtime to collect statistics on call times. (Disabled by default.)
- `--enable-old-cplusplus-support[=`*PROG*`]` -- enables old original ILU-style C++ support. This support is not compatible with the CORBA 2.0 style of mapping for C++. If *PROG* is specified, treats *PROG* as the command to use for C++ compilation; *PROG* must be the full path name. Default is to enable C++ support iff the environment variable *CXX* is defined, in which case *CXX* should be the full path name of the compilation command.
- `--enable-corba-cplusplus-support[=`*PROG*`]` -- enables CORBA-style C++ support. This support is currently broken on UNIX. This support is not compatible with the original ILU style of mapping for C++. If *PROG* is specified, treats *PROG* as the command to use for C++ compilation; *PROG* must be the full path name. (Disabled by default.)
- `--with-cplusplus-mapping=`*OPTION* -- only valid if **--enable-corba-cplusplus-support** is also specified. Controls which of the three major CORBA C++ mapping genres is produced, depending on whether your C++ compiler has support for namespaces, nested classes, or neither. *OPTION* must be one of `Namespaces`, `NestedClasses`, or `Underscores`. If not specified, the configuration process attempts to choose something reasonable.
- `--enable-java-support` -- enables Java support. (Default is to enable it if a program called `java` is on your `PATH` environment variable, and if the configuration script decides that the Java installation is well-formed; disabled otherwise.)
- `--with-java-nopoll` -- specifies that the Java runtime must use the `select` system call rather than `poll`. This is needed only on Linux, where we haven't yet figured out how to make autoconf make the right decision automatically.

- `--with-java-threads=KINDS` -- specifies which version(s) of Java's runtime multi-threading support to configure ILU for. Possible choices for KINDS are green, native, or both. Use this only if the default configuration logic gets it wrong. Don't use this if ILU is being configured to use JNI for native methods.
- `--enable-java-jni` -- specifies that the Java runtime must use JNI for the Java native method implementation. Use this only on Java releases jdk1.2 and later as ILU requires JNI features not available on previous releases. Default is: use JNI only if the older native method implementation is not available and the Java release is jdk1.2 or later.
- `--enable-java-with-omg` -- specifies that the Java runtime should includes ILU's own version of org.omg.CORBA classes. Default behaviour is to use ILU's classes with jdk 1.1 and the standard classes with jdk1.2.
- `--enable-java-cosnaming-prefix` -- force use of standard (but conflicting) CosNaming prefix for Java classes. (Disabled by default.)
- `--enable-python-support{=`*PYTHON-HOME*`}` -- enables Python support. If *PYTHON-HOME* is specified, treats *PYTHON-HOME* as the directory in which the Python system is installed. Default is: enabled if the program `python` is on your `PATH` environment variable; disabled otherwise.
- `--enable-old-python-gc-behavior` -- Prevents Python true objects from being garbage collected by the Python runtime, even if there are no Python references to the object. This was the default in versions of ILU previous to 2.0alpha10. The default is to disable this.
- `--enable-corba-python-mapping` -- Causes the Python stubber to produce, and the runtime to expect, the proposed CORBA mapping of OMG IDL (and ILU ISL) to Python, instead of the 'classic' ILU mapping. This CORBA mapping is a work in progress; we expect significant changes to it in the future. Use at your own risk. The default is to disable this.
- `--enable-python-dictionaries` -- In both the 'classic' and CORBA mappings of ILU ISL to Python, causes all sequence types matching a certain profile to be mapped to and from Python dictionaries instead of Python lists. The sequence type must have a name that ends with either `"`dict`"` or `"`Dict`"`; the base type of the sequence type must be a record type; the record type must have exactly two fields; the two fields must be named `name` and `value`; and the type of the `name` field must be either an integer, byte, string, or cardinal type. The default is to disable this.
- `--enable-perl-support{=`*PERL-HOME*`}` -- enables Perl support. If *PERL-HOME* is specified, treats *PERL-HOME* as the directory in which the Perl system is installed. Default is to enable this iff the PERL environment variable is defined (and in that case it should be defined to be the absolute filename of the Perl interpreter).
- `--enable-guile-support` -- enables Guile Scheme support. Default is: enabled if the program `guile` is on your `PATH` environment variable, disabled otherwise.
- `--enable-lisp-support[=`*PROG*`]` -- enables Franz Allegro Common Lisp support. Default is: enabled if any of the programs `franz`, `cl`, or `lisp` (in that order) are on your `PATH` environment variable, disabled otherwise. You can also explicitly specify the full pathname of the Lisp interpreter with *PROG*.

- `--enable-old-lisp-method-names` -- at one point, the mapping of method names to Common Lisp was `interface:methodname`. This was ambiguous; it was changed to `interface:typename.methodname`. If this switch is specified, both old and new names will be available. (Disabled by default.)
- `--enable-xview-support` -- builds the libraries to interface the XView GUI library to the ILU main loop. (Disabled by default)
- `--enable-xt-support` -- builds the libraries to interface Xt-based GUI libraries to the ILU main loop. (Disabled by default)
- `--enable-tk-support` -- builds the libraries to interface Tk-based GUI libraries to the ILU main loop. If this is enabled, the switches **--with-tk-includes-dir** and **--with-tcl-includes-dir** should also be specified. (Disabled by default)
- `--with-tk-includes-dir=DIR` -- specifies where to find the Tk header files.
- `--with-tcl-includes-dir=DIR` -- specifies where to find the Tcl header files.

The particular ANSI C compiler to use may be specified by setting the environment variable `CC` to the full path name of the C compilation command before running `configure`. Similarly, the particular C++ compiler to use may be specified by setting the environment variable `CXX` to the full path name of the C++ compilation command to use.

Once you've run the configure script, the output is stored in several files. The file which contains the symbols which control all of the Makefiles in the system is in *ILUSRC*/imake/ilu.defs.new. If you wish to fiddle with compiler options or things of that sort, that's the file to hack, before doing the make.

17.2.4.2 Manual Fixups for Threading

Sadly, our autoconf stuff is not yet fully up to the task of configuring for use of threads; you sometimes have to do a manual step or two, before and/or after running `configure`. This is better than it used to be; manual fixups should no longer be required for Solaris 2, OSF 3--4, IRIX 6.2--6.7, Linux 2.0, or AIX 4.1.4--4.4.

On some operating systems, linking POSIX threads programs requires a special flag, `-lpthread`, to appear at the end of the linkage command line. On others, the special flag is `-lthreads` or `-lpthreads`. If you operating system is not one of those listed above and you've configured with `--enable-os-threads`, you should find out out what the appropriate library for your system is and then edit '*ILUSRC*`/imake/ilu.defs.new`' after running `configure` but before running `make`. You should find a definition of the `make` variable `SYSAUX_LIBRARIES` and fix it (if necessary) to end with the appropriate `-l`*whatever* for your system's threads.

On some operating systems the C and C++ compilers require a certain preprocessor symbol be `#define`d when compiling sources to be included in threaded programs (and it's OK to `#define` these symbols for single-threaded programs too). If configuring to include OS-supplied thread support on an operating system not listed above but requiring such a symbol definition, make sure you also explicitly supply a C compilation command, and that it includes `-D`*whatever* to `#define` this symbol.

On Linux, when using Provenzano's pthreads (POSIX threads) library, you use special scripts provided instead of `gcc` and `g++`. Those scripts are normally located at '`/usr/local/pthreads/bin/pgcc`' and '`/usr/local/pthreads/bin/pg++`'. Use the facilities described above to configure these scripts as your C and C++ (if you're doing C++) compilers. It's OK to compile even single-threaded programs this way. On our Linux systems, these scripts produce the following warning messages when linking executables:

```
bfd assertion fail /opt/release/pub/bin/binutils/bfd/elf32-i386.c:624
bfd assertion fail /opt/release/pub/bin/binutils/bfd/elfcode.h:4716
```

Despite the dire-sounding warnings, the linker seems to produce working executables. Provenzano knows about this, but hasn't tracked it down yet. Sadly, the warning messages trick our autoconf script into thinking this compilation failed, and thus that the requested compilers aren't ANSI-C compliant. To cope with this, we configure to compile with scripts that call the Provenzano scripts and filter out these messages; here's the one for C:

```
#!/bin/sh -f
/usr/local/pthreads/bin/pgcc $* 2>/tmp/$$-cctmp
ccstatus=$?
fgrep -v "bfd assertion fail /opt/release/pub/bin/binutils/bfd/elf32-i386.c:624"
</tmp/$$-cctmp | fgrep -v "bfd assertion fail /opt/release/pub/bin/binutils/bfd/elfco
>&2
rm -f /tmp/$$-cctmp
exit $ccstatus
```

17.2.5 Building

Now that you have configured the release, do the following to build the system. Note that the capitalization of the arguments to `make` is important.

1. Set your working directory to *ILUSRC*:

   `% cd` *ILUSRC*

2. Build the system with the command:

   `% make`

3. You can then try a simple test with:

```
% cd ILUSRC/examples/test1
% make test
../../stubbers/c/c-stubber  Test1.isl
header file for interface Test1 to ./Test1.h...
common code for interface Test1 to ./Test1-common.c...
code for surrogate stubs of interface Test1 to ./Test1-surrogate.c...
code for true stubs of interface Test1 to ./Test1-true.c...
../../stubbers/c/c-stubber  Test2.isl
header file for interface Test2 to ./Test2.h...
```

```
common code for interface Test2 to ./Test2-common.c...
code for surrogate stubs of interface Test2 to ./Test2-surrogate.c...
code for true stubs of interface Test2 to ./Test2-true.c...
../../stubbers/c/c-stubber  Test3.isl
header file for interface Test3 to ./Test3.h...
common code for interface Test3 to ./Test3-common.c...
code for surrogate stubs of interface Test3 to ./Test3-surrogate.c...
code for true stubs of interface Test3 to ./Test3-true.c...
rm -f clnt.o
cc -c  -g -I. -I../../runtime/c -I../../runtime/kernel clnt.c
rm -f Test1-surrogate.o
cc -c  -g -I. -I../../runtime/c -I../../runtime/kernel Test1-surrogate.c
rm -f Test1-common.o
cc -c  -g -I. -I../../runtime/c -I../../runtime/kernel Test1-common.c
rm -f Test2-surrogate.o
cc -c  -g -I. -I../../runtime/c -I../../runtime/kernel Test2-surrogate.c
rm -f Test2-common.o
cc -c  -g -I. -I../../runtime/c -I../../runtime/kernel Test2-common.c
rm -f Test3-surrogate.o
cc -c  -g -I. -I../../runtime/c -I../../runtime/kernel Test3-surrogate.c
rm -f Test3-common.o
cc -c  -g -I. -I../../runtime/c -I../../runtime/kernel Test3-common.c
rm -f client
cc  -g -I. -o client  clnt.o Test1-surrogate.o Test1-common.o \
  Test2-surrogate.o Test2-common.o Test3-surrogate.o Test3-common.o \
  ../../runtime/c/libilu-c.a ../../runtime/kernel/libilu.a
rm -f srvr.o
cc -c  -g -I. -I../../runtime/c -I../../runtime/kernel srvr.c
rm -f Test1-true.o
cc -c  -g -I. -I../../runtime/c -I../../runtime/kernel Test1-true.c
rm -f Test3-true.o
cc -c  -g -I. -I../../runtime/c -I../../runtime/kernel Test3-true.c
rm -f server
cc  -g -I. -o server  srvr.o Test1-common.o Test1-true.o \
  Test2-common.o Test3-common.o Test3-true.o \
  ../../runtime/c/libilu-c.a ../../runtime/kernel/libilu.a
./testserver
Starting server...
Running client against server...
Client run successful.
Killing server...
./testserver: 27469 Terminated
Exiting with status 0.
%
```

4. If the build goes well, install the system with the command

   ```
   % cd ILUSRC
   % make Install
   ```

5. After the installation is complete, you may remove extra files in *ILUSRC* with the command

   ```
   % make Clean
   ```

You may wish to use `make Clean` at any time, to get your system into a consistent state.

6. If you change the configuration files, you should clean the system with the command '`make Clean`', and redo the installation starting at step 2. If you run into problems that can be fixed without changing the configuration files, you can re-build the system by starting at step 3.

17.2.6 Environment Variables

ILU tools use a number of UNIX environment variables under the covers. Note three distinct phases when these variables might have significance: (1) when building and installing ILU, (2) when developing an ILU application, and (3) when running an ILU applicaiton.

- `ILUHOME`: there are C preprocessor and make variables of this name, holding the value of *ILUHOME*. The environment variable of this name is rarely, if ever, significant. The only possibilities are in: phase 2 use of the LISP stubber, phase 3 use of LISP, and phase 3 use of Scheme.
- The environment variables `ILUPATH` and `ILUPATH_NO_ILUHOME` are significant to the stubbers and other interface-processing tools, normally used in phase (2). They collectively define a set of directories to be appended to the interface search path given on the relevant tool's command line. If `ILUPATH` is not defined, '`.`' and '*ILUHOME*`/interfaces`' are appended. If `ILUPATH` is defined, it should contain a colon-separated list of directories, and they are appended, followed --- unless `ILUPATH_NO_ILUHOME` is defined (with any value) --- by '*ILUHOME*`/interfaces`'.
- You may want your `PATH` environment variable to include the directory '*ILUHOME*`/bin`' during phase (2). This makes it possible to invoke ILU development tools (e.g., stubbers, TIM tools) by short names. Most of the ILU development tools can be invoked by their full pathnames, without '*ILUHOME*`/bin`' being on your `PATH`; some of the TIM tools require '*ILUHOME*`/bin`' to be on your `PATH`.
- After phase (1), your `MANPATH` variable can have the directory '*ILUHOME*`/man`' on it.
- If you are using Common Lisp, the portable DEFSYSTEM included with ILU uses the value of `SYSDCLPATH` to find system descriptions. It should be a colon-separated list of directories. A good initial value might be `.:${ILUHOME}/lisp`. See Appendix A of the reference manual for more details on the portable DEFSYSTEM.
- If you are using Python, in phase (3) the environment variable `PYTHONPATH` should include the directory in which the `ilu` library for Python has been installed; that's normally '*ILUHOME*`/lib`'.
- If you are using the language Java, make sure the `java` interpreter is on your `PATH`, and that '*ILUHOME*`/bin`' is on your `PATH`. The value of the environment variable `CLASSPATH` should have '*ILUHOME*`/lib/ilu.jar`' on it, or if on Windows, '*ILUHOME*`/lib/classes`' on it. Add the '*ILUHOME*`/lib/ilujavaobv.jar`' to the class path, if the program uses the Java object by value feature. The value of the environment variable `LD_LIBRARY_PATH` should have '*ILUHOME*`/lib`' on it.
- If you are using Guile Scheme, the value of the environment variable `LD_LIBRARY_PATH` should include the directory in which the `ilu` library for Guile has been installed; that's normally '*ILUHOME*`/lib`'.

Additionally, the environment variable `SCHEME_LOAD_PATH` should contain '*ILUHOME*`/guile`', so that the ILU Guile files can be found.

- During phase (2), the variable `ISLDEBUG` can optionally be set to any value to enable tracing in the ISL parser.
- During phase (2), the variable `ILU_TYPE_UID_VERBOSE` can optionally be set to any value to enable tracing the calculations of type UIDs in the stubbers.
- During phase (3), the variable `ILU_DEBUG` can be optionally be set to a colon-separated list of trace values to enable tracing in the ILU runtime kernel. See Section 16.7 [Debugging ILU Programs], page 236, for more information.
- If you are using the ''shared filesystem'' approach to ILU simple binding, the default directory for registration files may be overridden at run-time by setting the environment variable `ILU_BINDING_DIRECTORY` to the path of a different directory.
- If you are using the ''ILU service'' approach to ILU simple binding, the default values for the *REALM-NAME*, *HOST*, and *PORT* may be overridden by setting the value of the environment variable `ILU_BINDING_SERVICE` to a string of the form `"`*realm-name*`:`*host*`:`*port*`"`. Any of the three fields may be empty, so you could override just the port, for example, by using a value of the form `"::2034"`, which would mean to use the default values for REALM-NAME and HOST, and the value of `2034` for the port.
- Certain language runtimes, including the ANSI C, Java, and Python language runtimes, support the standard CORBA method `CORBA::ORB::list_initial_services()` during phase (3). If the environment variable `ILU_COS_NAMING_IOR` is set to the IOR of a CosNaming service, ILU will offer the `NameService` service, using that IOR to access the service.
- During phase (3), ILU can experience three kinds of internal error conditions: assertion failures, memory allocation failures, and 'check' failures (similar to an assertion failure). What it does when any of these three are experienced can be set, in the Python runtime, by setting the environment variables `ILU_ASSERTION_FAILURE_ACTION`, `ILU_MEMORY_FAILURE_ACTION`, and `ILU_CHECK_FAILURE_ACTION` to an integer value, which is then used to set the respective ILU kernel failure mode. See '*ILUSRC*`/runtime/kernel/iluxport.h`' for the documentation of which integer codes are appropriate for `ilu_SetAssertionFailureAction()`, `ilu_SetMemFailureAction()`, and `ilu_SetCheckFailureAction()`.
- During phase (3), in the Python runtime, the Python import mechanism is augmented by default with an additional module loader which will load support for ILU ISL or OMG IDL interfaces found on the `ILUPATH` environment variable directly into Python. Automatic enabling of this mechanism can be defeated by setting the environment variable `ILU_PYTHON_DISABLE_AUTOIMPORT` to any value before loading the ILU module into Python. In addition, setting the variable `ILU_PYTHON_IMPORT_VERBOSE` will cause the auto-import mechanism to print status messages when loading an interface.

17.2.7 Testing the Build

There are several ways to test the build. The most straightforward is to build and install it somewhere. Set up your environment variables as described above. Then make a scratch directory, which we'll refer to as *TESTDIR*, and do the following:

```
% cd TESTDIR
% cp ILUHOME/examples/test1/* .
% ilumkmf
% make client server
ILUHOME/bin/c-stubber  Test1.isl
header file for interface Test1 to ./Test1.h...
common code for interface Test1 to ./Test1-common.c...
code for surrogate stubs of interface Test1 to ./Test1-surrogate.c...
code for true stubs of interface Test1 to ./Test1-true.c...
ILUHOME/bin/c-stubber  Test2.isl
header file for interface Test2 to ./Test2.h...
common code for interface Test2 to ./Test2-common.c...
code for surrogate stubs of interface Test2 to ./Test2-surrogate.c...
code for true stubs of interface Test2 to ./Test2-true.c...
ILUHOME/bin/c-stubber  Test3.isl
header file for interface Test3 to ./Test3.h...
common code for interface Test3 to ./Test3-common.c...
code for surrogate stubs of interface Test3 to ./Test3-surrogate.c...
code for true stubs of interface Test3 to ./Test3-true.c...
rm -f clnt.o
cc -c -g -I. -IILUHOME/include  clnt.c
rm -f Test1-surrogate.o
cc -c -g -I. -IILUHOME/include  Test1-surrogate.c
rm -f Test1-common.o
cc -c -g -I. -IILUHOME/include  Test1-common.c
rm -f Test2-surrogate.o
cc -c -g -I. -IILUHOME/include  Test2-surrogate.c
rm -f Test2-common.o
cc -c -g -I. -IILUHOME/include  Test2-common.c
rm -f Test3-surrogate.o
cc -c -g -I. -IILUHOME/include  Test3-surrogate.c
rm -f Test3-common.o
cc -c -g -I. -IILUHOME/include  Test3-common.c
rm -f client
cc -g -o client clnt.o Test1-surrogate.o Test1-common.o \
  Test2-surrogate.o Test2-common.o Test3-surrogate.o \
  Test3-common.o  ILUHOME/lib/libilu-c.a \
  ILUHOME/lib/libilu.a
rm -f srvr.o
cc -c -g -I. -IILUHOME/include  srvr.c
rm -f Test1-true.o
cc -c -g -I. -IILUHOME/include  Test1-true.c
rm -f Test3-true.o
cc -c -g -I. -IILUHOME/include  Test3-true.c
```

```
rm -f server
cc -g -o server srvr.o Test1-common.o Test1-true.o \
  Test2-common.o Test3-common.o Test3-true.o \
  ILUHOME/lib/libilu-c.a ILUHOME/lib/libilu.a
% ./server &
[1] 7079
% exported ilu:Test1-Server/Test1_Initial_Object;ilu%3AiX2w6hjR-...
% ./client
Test1.O1.U-CSS-to-U
u._d=5, u._u.boolean = 1, u._u.O1 = 0x1ffee7c
Test1.O1.f-CSS-to-R0
ro->i=9
Test1.O1.R-ScS-to-F
f=39.700001
Test1.O1.a-RO
Test1.O1.get-O2
got O2, sbh = ilu:Test1-SunRPC-Server/1;ilu%3AaUtts57Ywbp2fxe6+-...
Test1.o2.OO-A0-to-CSS
Test1.O2.R-I-A1-to-I-A0
Test1.O1.get-O3
making O3...
got O3, sbh = ilu:Test1-Server/2;ilu%3An+eRrvAZ8JB9v2qoX7sJGPxdX...
Test1.O3.RS-R-to-R-IS
Test1.O3.O1-U-to-U
u._d=3, u._u.boolean = 0, u._u.O1 = 0xd2b78
Test1.O1.get-O3
got O3, sbh = ilu:Test1-Server/3;ilu%3Ab-mNa9uj0TsJAp7YrlEh0AUfX...
Test3.O.RS-R-to-R-IS
Test3.O.O1-U-to-U(0xd7520, {3})
u._d=3, u._u.boolean = 0, u._u.O1 = 0xd2b78
Test3.O.I-to-Test1U(397)
Test3_O_I_to_Test1U:  u2._d=5, u2._u.boolean = 1, u2._u.O1 = 0x10a88d0
Test1.O1.get-O3
making O4...
got O3, sbh = ilu:Test1-Server/4;ilu%3Ad8sZGQLLpVsJ2PBL5BoIX45qO...
Test1.O4.R_to_R (12345.6789000000) => 1020304.0506070800
doubles:  r1 is 12345.6789000000, r2 is 1020304.0506070800
%
```

You can proceed to test the various other clients and servers in different languages against each other. See the file '*ILUHOME*`/examples/test1/README`' for more information.

17.2.8 Notes on Specific Systems

17.2.8.1 AIX 4.2

From Yongjun Zhang, `zhang@quickturn.com`: ''When using plain xlc on AIX 4.2, when configured with support for OS threads with --enable-os-threads, my images would SEGFAULT. Switching to the xlc_r compiler fixed that.''

17.2.8.2 HP/UX

From `hassan@db.stanford.edu`: ''In order to get ILU 2.0a to compile on HP/UX, I had to set the CC environment variable to the following before running configure: `setenv CC "/bin/cc -Aa +z -D_HPUX_SOURCE"`.''

17.2.8.3 DEC ALPHA with OSF OS

From `hassan@db.stanford.edu`: ''Use `cc` instead of `gcc` as the C compiler, and make sure to include the '-taso' switch.''

From `jg@w3.org`: ''I built [ILU 2.0 alpha on OSF 3.2B] without the '-taso' switch. Is this still needed? `c-stubber` certainly ran without it this release.''

17.2.8.4 SunOS 4.1.x

Note that the default Sun C compiler is not ANSI C, nor is `gcc` when installed against the normal Sun header files and '`/lib/libc.a`'. You will have to use either `gcc` with the GNU C Library glibc, or the SunPro ANSI C compiler `acc`, or Lucid Energize `lcc`, or some other ANSI compiler.

17.3 Examples

The following example uses of ILU are provided in the installed tree as subdirectories of '*ILUHOME*`/examples/`' (those of any given language are listed roughly in order of increasing complexity/obscurity):

- '`timeit`' -- a simple, crude timing test. In C, and one of our simplest examples in that language.
- '`dialog`' -- This simple example pauses for user input everywhere the application gets control. By deliberately breaking network connectivity at one of these pauses, one can test ILU's reaction to network failures. Available only in C, and another of the simplest examples in that language.
- '`cubit`' -- our simplest example in Python, using a simple OMG IDL interface designed by Sun and contributed as part of their free IIOP implementation. If you use the Sun tools to build their clients and servers, they should interoperate with the ILU-built servers and clients.
- '`cpp2foo`' -- a very simple example used to demonstrate ILU's CORBA 2 C++ stubber and runtime. Available for UNIX and Windows.
- '`test1`' -- this is a rambling, random example, which serves as a basic informal regression test. It uses one client program and one server program, and tests a variety of basic features and data types.

The clients and servers are built for: C, C++, Python, Java, Common Lisp, and Guile (or whichever of those you have configured ILU to support); they should interoperate in all combinations. If you are interested in working with Common Lisp, or our old C++ support, or Guile Scheme support, we recommend starting here.

- '`javatest1`' -- the simplest possible Java example for our new Java support.
- '`tutorial`' -- this contains the code for a small ILU tutorial, implemented in Java, ANSI C and Python. If you are interested in using either of these languages with ILU, we suggest starting with the code in this directory, and one of the files '*ILUHOME*`/doc/tutorial-Java.ps`', '*ILUHOME*`/doc/tutorial-C.ps`' and '*ILUHOME*`/doc/tutorial-Python.ps`'. Even if you want to use a different programming language, it's probably worth your while reading one of these tutorials.
- '`multlang`' -- an example of using multiple languages (ANSI C and Python) in the same address space, communicating via ILU.
- '`changeup`' -- a test of closing servers and connections, and of recovery from such things; used in our informal regression testing. Available only in C.
- '`objtable`' -- a C example of the use of an object table, to create true instances on the fly when some client uses them.
- '`black-widow-bank`' -- a tiny banking example in Python, based on the VisiBroker for Java "bank" example. Should interoperate with the VisiBroker for Java "bank" example.
- '`orbix-grid`' -- a simple example in Python, based on the Orbix "grid" example. Uses OMG IDL and IIOP.
- '`orbplus-bank3`' -- a modest example in Python and LISP, based on the HP OrbPlus "bank3" example. Uses OMG IDL and IIOP.
- '`httest`' -- three programs that test and demonstrate the use of the HTTP protocol within ILU. They show ILU communicating with an existing Web server, an existing Browser communicating with ILU, and ILU communicating with ILU over HTTP. All in C.
- '`iiop`' -- a simple example that exercises the IIOP support in ILU, according to the test pattern originally developed by David Brownell for the Sun IIOP example code. All in C.
- '`pickle`' -- a simple example that exercises the pickle support in the IIOP protocol, mainly via Python. You must have Python support to use this test, though there is also a C server, which can be used with Purify to test memory usage of pickle code, and a Java server which is good for nothing except testing Java.
- '`test2`' -- a collection of tests of non-basic features, used in our informal regression testing. The collection consists of: a test of concurrent protocols, a test of `ASYNCHRONOUS` methods, a test of pipelining, and a test of call order preservation (AKA serialization), and a test of both pipelining and call order preservation. All are available only in C.
- '`javaserial`' -- a simple example that demonstrates support for "full custom" mapping in ILU. All in Java.

- ‘blob’ -- a contributed Python example that uses Tk.

Read the ‘README’ file in each directory first.

17.4 Name Servers

No standard ‘‘name service’’ or binding service is provided. We feel that this is an area to be addressed independently, and we may include a name service in future releases of ILU. An experimental simple name service bootstrap interface is available as the *simple binding system*. See the ANSI C ILU_C_PublishObject, ILU_C_WithdrawObject, and ILU_C_LookupObject, and corresponding routines in the other languages, for more details. This interface is intended to be only sufficient to find the real name service.

Two implementations of this are available, one using an ILU service to store the information, the other using a shared filesystem. They can be selected at configuration time, by specifying either "--with-binding-dir=DIRECTORYNAME", or "--with-binding-service=REALM:HOST:PORT", where REALM may be a user-specified string identifier, that is the name of some conceptual space which the simple binding server serves. These values are compiled into the ILU kernel library, but may be overridden with environment variables at runtime.

17.5 Documentation

ILU documentation is provided in a pre-formatted form, PostScript. The source form of the documentation is called TIM, and is documented in the ILU reference manual. If for some reason you do need to rebuild the documentation, you should have the systems TeX, Perl, ghostscript, dvips, and pbmplus; if you can’t find these yourself, please send mail to ilu-core@parc.xerox.com for info on how to find them.

17.6 Mailing Lists

To be added to, or deleted from, any ILU mailing list, please send mail to ilu-request@parc.xerox.com. **Do not** send mail to the list itself.

The general ILU discussion mailing list is ilu@parc.xerox.com. People post questions, discuss changes, and help each other out on that list. Another list, used only for announcements of ILU things, and consequently much lower-volume, is ilu-interest@parc.xerox.com. The ilu list receives everything that the ilu-interest list receives; there is no need to be on both lists. Again, send mail to ilu-request@parc.xerox.com to be added to or removed from either of these lists.

Archives of these lists can be found at http://www.findmail.com/listsaver/ilu/?archive/.

17.7 Changes

17.7.1 Changes from 2.0alpha13 to 2.0alpha14

- **New Type UID Hash Algorithm.** This release introduces a documented algorithm for producing the ILU 'type hash' for an ISL type. It's documented in the ILU manual, in the section entitled ''Algorithm for Generation of Structural-Hash Type IDs''. This algorithm will become the default algorithm for the next release, but the old algorithm is still the default for this release. You can enable the new algorithm with the configuration switch **--enable-version-2-type-uids**. We recommend that you do this, if possible. ILU configured with the version 2 type uids will *not* be on-the-wire compatible with ILU configured with the old type uids.
- **Directives in ISL.** The ISL syntax has an experimental directives concept. Expect changes before this is made final.
- **Java support**. Supports running ILU and a standard CORBA ORB in the same address space. Optional support for JNI based native methods. Support for jdk1.2. Support for Microsoft sdk3.1. Switchable usage of org.omg.CORBA classes (Java core for jdk1.2 or provided by ILU for jdk1.1). No more problems with the boot class path. Configuration process slightly improved.
- **Java API changes**. Default use of org.omg.CORBA.Object interface as base type for all ILU stubbed objects. Corba system exceptions are no longer subclassed; other minor API changes required for sharing address space with standard ORBs. Most users shouldn't notice these changes.
- **Java serialization support**. Most Java classes and interfaces corresponding to ISL types are now serializable. Among other things, this allows ILU objects to be passed via `Java RMI`. Also, a new ILU interface exists which allows transport of arbitrary Java serializable objects with ILU. Third, a very experimental `DIRECTIVE-EXPERIMENTAL` mechanism which optionally allows classes implementing ILU object types to avoid extending org.omg.CORBA.Object (and optionally, extend java.rmi.Remote). Also, custom mapping may now be specified in the ISL file using the `DIRECTIVE-EXPERIMENTAL` keyword.
- **Support for Python CORBA mapping.** This release includes a preliminary version of the CORBA mapping for Python. You should configure ILU with **--enable-corba-python-mapping** to get this form of Python support, instead of the ''classic'' version documented in the ILU reference manual. See `http://www.informatik.hu-berlin.de/~loewis/python/pymap.htm` for more information on this mapping.
- **Python has experimental support for passing dictionaries.** If **--enable-python-dictionaries** is specified during configuration, the mapping of ISL to Python generated by the stubber will be modified as follows: If a sequence type has a base type which is a record type with two fields, `name` and `value`, and the type of the `name` field is an integer or string type, and the name of the sequence type ends with `"dict"` or `"Dict"`, the sequence type will be mapped to a Python dictionary instead of to a list. The key value of each item in the dictionary will be the value of the `name` field of a record value, and the value value of the item will be the value of the corresponding `value` field of the same record value. This mapping may change in the future.
- **Python 'auto-import' of ILU interfaces now enabled by default.** In this release, loading of the Python `ilu` module will cause a call to `ilu.AutoImport()`, with no parameters. This can be

defeated by setting the environment variable `ILU_PYTHON_DISABLE_AUTOIMPORT` before loading the `ilu` module, and `ilu.AutoImport` can still be called manually at a later time.

- **Lisp "old-style" method names not produced by default.** The default for lisp is to not produce the old-style method names. Users who need them should specify the **--enable-old-lisp-method-names** configuration switch to get them.

17.7.2 Changes from 2.0alpha12 to 2.0alpha13

- **Clarified copyright**. We have clarified the terms of the ILU copyright to make it clear that it conforms to the requirements of free software.
- **Java support more robust**. Support for different java environments more robust. Added JDK1.2betaX. Added support for native threads on unix. On NT, supports developping ilu applications from within a few commercial IDE's. Many bug fixes.
- **GSS security transport**. This provides the ability to wrap arbitrary security contexts around communication between two address spaces. It includes the ability to identify callers by arbitrary GSS namespaces schemes. A generic GSS *shell* (into which various mechanisms can be plugged) is included. See the security chapter of the ILU manual for more information.
- **More CORBA-ization of the C runtime**. The pseudo-ORB initialization functions are now present for the C runtime.
- **Conforming implementations of the HTTP-NG wire protocol and MUX transport**. Implementations of the HTTP-NG wire protocol and webmux transport now conform to the Internet drafts for these protocols. Note that the mux transport only works in threaded mode (both client and server).
- **Proper sending of clean shutdown messages**. It is now possible for RPC and transport protocols to send clean shutdown messages. Currently only the HTTP-NG wire protocol, `w3ng`, does so.
- **Reaping idle incoming connections on multi-threaded servers**. It used to be the case that incoming connections to multi-threaded servers would not be closed when the server was trying to reduce its FD (File Descriptor, an OS-level resource) usage; this has now been fixed.
- **XML parser**. This release includes an early version of a C-based validating XML parser. It is a general-purpose XML parser that may be used in future releases of ILU for various purposes. Information on the use of the parser and its current (known) limitations is found in '*ILUSRC*`/stubbers/XML-parser/README`'.
- **Optimized marshalling of data structures from C**. The C marshalling code for ILU now matches data structures in memory against their marshalled representation in the particular wire protocol selected. If they match, the data structure is written directly to the wire, instead of having each element manipulated independently. This can speed up the I/O processing of a large array of floating-point values, say, by over 2000 percent.
- **Bug fixes in Python and Java stubbers may cause type UID changes**. If you stub multiple files on the same command line with the Java and Python stubbers, past releases of ILU may have given different type UIDs to the resulting types than if you stubbed each file separately. This bug has

been fixed, and the Python and Java stubbers now always give the same type UID as the other stubbers produce.

- **Better configuration support for Linux**. ILU should configure 'out of the box' properly for Red Hat 5.1 Linux. In general, updates to the configuration machinery should make Linux configuration easier.

17.7.3 Changes from 2.0alpha11 to 2.0alpha12

- **Many bug fixes**. This release primarily fixes a number of bugs in 2.0alpha11, including various problems with the direct IDL support, the Common Lisp support, the 'serializer' construct in the kernel, and the new CosNaming name service.
- **First release of IETF Generic Security Service API shell.** This is an implementation of a 'shell' library providing the API defined by Internet RFC 2078 (`http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2078.txt`), whichprovides a generic interface for providing security contexts around communications between parties. The ILU GSS shell provides a 'back-end' API, into which specific namespace schemes and authentication mechanisms can be 'plugged'. It also includes one example of a (non-secure) namespace scheme (called `rfc822`), and one example of a (non-secure) authentication mechanism (called `nil`). Due to U.S. export controls on cryptographic technology, we don't plan to release publicly any 'secure' namespace or authentication mechanisms, as all of those that we're aware of rely on controlled cryptographic algorithms to provide that security. We expect that serious users of the GSS will provide their own namespaces and mechanisms. See '*ILUSRC*`/GSS/README`' for more information on this GSS shell. Note that this release does *not* provide support for the ILU `gss` transport filter.
- **Support for the w3ng wire protocol.** This release includes support for a new wire protocol called `w3ng`. This is the first wire protocol actually designed for use with ILU. It exhibits a number of efficiency improvements over existing wire protocols supported by ILU.
- **Java support now includes 'full custom mapping'.** This allows a Java programmer to specify non-standard Java classes to be used for any ILU ISL constructed type. See the Java chapter of the manual to find out how to use this.
- **Server relocation supported.** It is sometimes useful to have a 'dummy' server, that will redirect any requests to it to a real server somewhere else. This can be used for load balancing, automatic start-up of services, redirecting name service, code migration, and other various purposes. ILU supports this via a mechanism called *server relocation*, currently available only in the C, Python, and Java runtimes, and only with the `w3ng` and `IIOP` protocols.
- **TCP affordances.** It is now possible to control the size of TCP/IP transport buffers, either as an optional parameter in the tinfo when creating a Port, or by setting the default buffer size directly from C or Python. It is also possible to get some elementary statistics about TCP/IP resource usage from either C or Python.
- **Simple use of ISL types in Java.** ISL types of any kind except `PICKLE` no longer automatically

interact with the ILU runtime, so the ILU runtime library for Java no longer needs to be loaded to support non-ILU use of these types.

- **Mechanism for discovering caller identity in C has changed.** You should now call `ILU_C_CallerIdentity()`, instead of `ILU_C_GetPassportContext()`, in a true method to determine the identity of the caller.

17.7.4 Changes from 2.0alpha10 to 2.0alpha11

- *Documentation re-done.* The documentation for the various language runtimes has been re-done to a standard template.
- *New OMG IDL parser.* A very nice parser for OMG IDL has been contributed by Martin von Loewis, and has been incorporated. It consists of two C files which are integrated into the parser library, instead of the huge and buggy C++ idl2isl program previously supplied (and no longer part of the distribution). Many thanks, Martin!
- *New ISL to IDL program.* The file '`stubbers/pprint/isl2idl`' converts isl2idl when possible.
- *Guile Scheme support.* Bill Nell at Siemens Corporate Research, Inc, has contributed the beginnings of support for Guile Scheme, including a Scheme stubber, a working ILU LSR for Guile, and working versions of the '`examples/test1/`' example client and server programs.
- *Multiple ports on server.* It is now possible to add multiple ports to an ILU server, so that the same set of objects can be exported via multiple protocols. All languges provide hooks to add multiple ports to a server.
- *Serialization and pipelining on non-concurrent protocols.* The ability to have calls delivered to the callee in serial order over a single connection has been added for non-concurrent protocols with an abstraction called *serialization*. In addition, the ability to have multiple outstanding calls on connection that uses a non-concurrent protocol has been added via an abstraction called *pipelining*. Access to this kernel functionality is currently only provided via the C LSR.
- *Custom surrogate support added for all languages.* The ability to specify a user-implemented class for surrogate instances has been added to all languages. This allows implementation of custom caching strategies and similar techniques.
- *Custom record support added for all languages.* The ability to override record type marshalling in languages where the type system allows it has been added. This capability is useful for support of objects-by-value. Not yet available in Common Lisp or Guile.
- *HTTP support improved.* The ILU `HTTP` support will now properly redirect for responses. It also responds to non-1.0 versions of the `HTTP` protocol more kindly, and has the beginnings of support for `HTTP 1.1`.
- *IIOP improved.* The IIOP support will now respond appropriately to relocation replies, common with other ORBs, though not used in ILU. It now supports CORBA 2.1 character set negotiation. A non-concurrent version of `IIOP` is available as `"siiop"`; note that since the OMG spec clearly identifies `IIOP` as concurrent (in the ILU sense), `"siiop"` should only be used between ILU clients and servers.

- *World Wide Web MUX transport included.* This release includes an implementation of the MUX transport described in `http://www.w3.org/Protocols/MUX/WD-mux-961023.html`,but with the modifications suggested in `http://lists.w3.org/Archives/Member/w3c-mux/msg00039.html`.This transport, called `"w3mux"`, allows for multiple simultaneous sessions or connections over a single TCP/IP connection. It supports message fragmentation and interleaving of streams. It supports bi-directional connections over the TCP/IP connection, thereby supporting callbacks through firewalls.
- *Java improvements.* Java support for Win32 platforms (using Javasoft JDK 1.1.x and Microsoft Java SDK 2.0 beta 2) is now included. The ILU support no longer works with JDK 1.0, but now supports JDK 1.1.x. Lots of minor improvements have been made. The generated stubs are now by default (mostly) compatible with the original Java mapping document (`ftp://ftp.omg.org/pub/docs/orbos/97-03-01.pdf`). The originalILU mapping is available as a stubber option.
- *CORBA 2.0 C++ support - limited* CORBA 2.0 C++ is currently working for Visual C++ (sans nested modules and pickles/anys support). [This is very new software - it has received very limited testing.]
- *CosNaming service included.* The distribution now includes an implementation of the CORBA name service, under '*ILUSRC*`/etc/CosNaming/`'. It is automatically built and installed if OMG IDL and IIOP support are configured in.
- *Sun RPC portmapper support included.* A C library for use with the Sun RPC `portmapper` is now included, in '*ILUSRC*`/etc/portmapper/`'. Included is an ILU ISL description of the `portmapper` interface, which can be used directly with other languages.
- *Auto-stubbing of modules supported for Python.* The `import` mechanism in Python can now be augmented by calling the function `ilu.AutoImport()`; this modifies `import` so that, when attempting to import a module called `foo`, if it fails to find '`foo.py`' and '`foo.pyc`' on the `PYTHONPATH` environment variable, it will then walk down the directories listed in the `ILUPATH` environment variable, looking for either '`foo.isl`', or, if OMG IDL support is enabled, '`foo.idl`'. If either of these are found, it will run the Python stubber on the file, putting the results in a temporary directory, and then load the resulting surrogate-side '`foo.py`' file.
- *FUNCTIONAL caching is now supported for Python and Lisp.*

17.7.5 Changes from 2.0alpha9 to 2.0alpha10

- *Default garbage collection behavior of Python true objects changed.* In previous ILU releases, the ILU kernel held an extra reference to each Python true object, so that they were never garbage collected. This extra reference has been removed in 2.0alpha10, so that the application must be careful to maintain references to objects which it wishes to preserve.
- *Aggressive garbage collection of C objects.* C objects (`ILU_C_Object *`) are now reference counted. An application must be careful to use `CORBA_Object_duplicate` and `CORBA_Object_release` correctly to avoid memory smashes.
- *Full type information cached.* If both `--enable-pickle-support` and `--enable-corba-iiop` have been selected, full type information on all compiled-in or dynamically-loaded ISL types is now cached in memory. This makes it theoretically possible for someone to write a CORBA Interface

Repository service for ILU (or something more useful). Note changes in `ilu_DefineMethod`, `ilu_DefineException`, and the new function `ilu_DefineMethodArg`.

- *HTTP persistence supported.* The `HTTP` protocol may now be selected with the string `"http_1_0p"`, which causes it to send `Connection: "Keep-Alive"` headers, and not close the connection between calls (assuming of course that the other end of the connection also supports this behavior - fairly common.) The programs in examples/httest have been updated accordingly. In addition, it is now possible to use `HTTP` over a boundaried transport.
- *OMG IDL exceptions with values handled.* The `idl2isl` compiler now, for an OMG IDL exception `E`, generates an ISL exception called `E`, and an ISL type called `ilu--prefix-idlExceptionType-E`. The stubbers handle this type variously; the C stubber renames it to `E`, as required by the CORBA spec; the Python stubber renames it to `E__omgidl_exctype`; the Lisp stubber folds it into the definition of the `condition E`, and doesn't support the type directly at all.
- *Java support improved.* The Java support has been improved, and brought closer to the emerging CORBA specification for it. Pickles are now supported, and work with IIOP; enumerations are now mapped according to the CORBA standard; system exceptions are now Java runtime exceptions; interfaces can be specified in OMG IDL; works with select-based (BSD) systems as well as poll-based (SVR4) systems; holder classes can be mapped the OMG way; many bug fixes.
- *Common Lisp support improved.* The Common Lisp work by Joachim Achtzehnter has been incorporated, and various other fixes have been added, including PICKLE support. Common Lisp support is still missing type registration, but in other respects should be fully working.
- *ilu_Server leaks fixed.* In previous versions, ILU kernel servers which had become empty were not garbage collected. This has been fixed. The fix also changes the `iluMainLoop` class in the old C++ runtime.
- *Default protocol and transport selected dynamically.* The default protocol and transport are now selected dynamically, so that ILU installations without `Sun RPC` can be created.
- *idl2isl now provided on Win32.* The `idl2isl` is now part of the Windows build.
- *Python 1.4 now provided on Win32.* Python 1.4 is now the version used on Windows systems.
- *WINIO no longer part of release.* WINIO, a subsystem no longer needed by ILU on Windows, but included in previous releases, has been dropped from the release distribution.

17.7.6 Changes from 2.0alpha8 to 2.0alpha9

- *PICKLE support for dynamic types added.* This allows you to use a new ISL type, `PICKLE`, to pass arbitrary typed values across interfaces. Pickle support is implemented in such a fashion as to be interoperable with CORBA `any`, and our OMG IDL to ISL translator in fact maps `any` directly to `PICKLE`.
- *Proper collection of dead connections.* The kernel now properly frees connections after they have been closed.
- *Prototype Java support.* This release contains a first pass at Java support for ILU.

- *Prototype Common Lisp support.* This release contains a first pass at Common Lisp support for ILU 2.0. Many thanks to Joachim Achtzehnter for contributing it!
- *Prototype Guile Scheme support.* Bill Nell and Siemens have been kind enough to contribute preliminary support for Guile, the GNU variant of Scheme. This contribution is provided ''as is'', in the '*ILUSRC*`/contrib/siemens-guile/`' subdirectory.
- *Support for Python 1.4.* A number of patches are incorporated to make ILU support for Python with Python 1.4 build ''out of the box'', on Unix. Python 1.3 is still the supported system for Windows.
- *Support for Xt fixed.* The support for using ILU with Xt now works.
- *Simplification of Windows support.* The Windows build has been simplified. WINIO has been removed, and everything builds with one set of makefiles. Support for WIN16 has been removed.

17.7.7 Changes from 2.0alpha7 to 2.0alpha8

- *HTTP protocol added.* This allows you to use the standard World Wide Web `HTTP`, version 1.0, between address spaces. This makes ILU programs Web servers and clients, though only in a very limited sense. See the 'Protocols and Transports' chapter of the user manual for more information.
- *OS threading added.* We've added support for use of the operating system's threads, if available, with the languages C, C++, and Python. The threading systems supported are POSIX threads, Solaris 2 threads, and Windows/NT threads. See '`examples/test1/srvr.c`' for an example of using threads.
- *ILU Simple Binding via an ILU service.* You can now choose to 'simple binding' via either a shared file system or via an ILU service. See the chapter on 'ILU Concepts' for more information.
- `ilusbls`. A program, `ilusbls`, that will list the objects known to the simple binding service, is provided. `ilusbls` will work with either the shared files simple binding or the ILU service simple binding.
- *Identities exported via a meta-object protocol.* Application-specific identity types may now be registered with the ILU kernel, and procedures to 'pickle' and 'unpickle' them are supported. Application-specific protocols and transports have access to these identity objects, and can use them for various security, accounting, and authorization strategies. See '`runtime/kernel/iluxport.h`' for more information on `ilu_IdentityInfo`.
- *Python threading supported.* If ILU is configured with `--enable-os-threads`, and your Python installation has been built with thread support, thread support will also be available in the Python ILU runtime. A new Python function, `ilu.ThreadedOperation()`, has been added to enable use of threads.
- *ILU/Python support for Windows NT.*
- *Change in default marshalling of discriminant references.* We switched to using a more efficient representation for server IDs when marshalling the discriminant of a call on the wire. This changes our ONC RPC and XNS Courier wire formats; the program numbers used have been changed to reflect this.

- *Change in algorithm to compute type UIDs.* We switched to a much more efficient algorithm for computing the structural fingerprint of a type. As a result, all stubbed files should be re-stubbed; the type IDs from ILU 2.0alpha7 will not be compatible with those of 2.0alpha8.
- Various fixes, to all the problems reported in `ftp://ftp.parc.xerox.com/pub/ilu/2.0/2.0alpha7-patches.html`, and more.

17.7.8 Changes from 1.8 to 2.0alpha

This release contains some major changes, and is NOT compatible "on the wire" with any previous version of ILU. There are also a few API changes. There may be further changes in 2.0beta and 2.0.

- We now use GNU autoconf (and still use imake).
- Support for C and C++ use on Windows 95 and Windows NT (Windows 3.1 coming soon), thanks to Dan Larner. Windows binaries are available (as well as source code).
- Thanks to Bridget Spitznagel, we now have support for cross-language calls within the same address space. Because we're not a compiler vendor, and can't keep up with all the compiler vendors in the world (not to mention all the combinations of them), we don't solve your problem of getting multiple language runtimes to co-exist. But where you *have* solved that problem (perhaps because you've got an easy instance, such as C and XXX), you can now just call through an ILU interface --- rather than having to write messy "foreign function" interfaces from one language to another. Each part of your program looks mono-lingual and normal, and we provide the control-flow and data-conversion glue to put them together. Data conversion is currently done by serializing and de-serializing to/from a normalized form in a memory buffer; we plan to investigate more direct methods (but not necessarily for release 2.0).
- Our "transport" abstraction has been re-organized. Among other things, this makes it (relatively) easy to introduce "filters" at the transport level. Of course, ILU remains open and extensible in this regard. Want to add a compression filter? Go ahead!
- ILU string binding handles become IETF URLs.
- We've made it possible for a calling application to interrupt a call in progress.
- The documentation (and of course, TIM) has diagrams and URLs!

  8) Generalized cleanup and bug fixing. This includes more attention to making it practical for others to add transport and protocol meta-objects. This also includes a more rigorous treatment of exceptions in the kernel and runtimes, with a taxonomy of exceptions aligned with CORBA's. It also includes fixes that change the type ID's and protocol mappings, which caused us to bump the major version number. All ONC RPC and Courier program numbers, and ISO object IDs are now official. The Courier type-ID-to-program-number mapping registry has been eliminated.

17.7.9 Changes from 1.7 to 1.8

- A kernel memory leak caused by having many clients connect to, then drop, a server was fixed.

- File descriptors are now removed from the event loop registry when a connection is closed, which fixes some errors in various runtimes.
- The kernel routines ilu_ConsiderSBH() and ilu_ReLookupObject() are now provided to change the binding of a surrogate kernel server. This allows a client to track changes caused when a server goes down and is re-started with different contact info. This should also handle the relocation requirement of CORBA's IIOP. They have not been fully tied into the language runtimes yet. Some language runtime code may still improperly keep a cache of an earlier SBH.
- The Common Lisp garbage collector is now tied in to the ILU network GC scheme, so that client interest in collectible true objects is communicated and used properly between Lisp clients and servers. Collectible true objects are now GC'ed.
- A new appendix to the ILU manual documents the process of adding ILU support for a different variety of Common Lisp.
- Various fixes to the Python support have been made to fix various bugs, and to allow unregistration of Tk event handlers when connections are closed. Python true objects must still be manually held onto by the server.
- References into freed data structures have been fixed in the C and C++ runtime, thanks to Purify.
- An authentication framework has been added, but no protocols currently pass any identity information except for the Sun RPC protocol's default authentication of "AUTH_UNIX". This identity is now available in C true method code, but the access method is not yet documented, as it will surely change.
- An obscure bug in the Lisp generic process code, responsible for causing an occasional "Bad Process-Lock" message, has been fixed.
- The XView X toolkit code in ILUSRC/etc/xview/ has been successfully used.
- Memory leaks in C true stubs have been fixed, and C true stubs now report unexpected exceptions properly.
- Fixes from hassan@db.stanford.edu for the DEC Alpha system with OSF/1 have been incorporated.

17.7.10 Changes from 1.6.4-p9 to 1.7

- The way of associating a Sun RPC (program number, version) tuple with an object type has changed. In release 1.6.4, the (program number, version) was assigned either manually or automatically, and a file maintained a list of (type ID, program #, version) tuples. Each client and server consulted this file when mapping between Sun RPC program #'s and ILU type ID's. This led to a number of problems. This scheme has been changed in release 1.7 to a scheme in which the Sun RPC program # is always the value 0x31000400, and the (32-bit) Sun RPC version is computed from the ILU type ID, using the CRC-32 hash algorith. Thus the version number is the CRC-32 of the ILU type ID. This has been tested for collisions, and they have been found to be extremely rare -- much rarer than collisions would have been under the ILU 1.6.4 scheme.

  This means that if you wish to use ILU 1.6.4 clients or servers with ILU 1.7, you should edit the 1.6.4 SunRPCRegistry file to use the 1.7 program number and version for each particular object type.

- The C runtime now offers an interface to threads, so that C servers can handle requests in different threads. This has been tested with the PPCR implementation of POSIX threads.
- Untested pseudo-threads libraries for the Xt and XView X toolkits are provided, under ILUSRC/etc/{Xt,xview}/.
- Support for the Python programming language has been added.
- All languages now support IN, OUT, and INOUT method parameters.
- Support for CORBA NIL object references has been added, via the new OPTIONAL keyword on object types. All object types defined with OMG IDL will be tagged automatically with OPTIONAL; object types defined with ILU ISL have the option of being OPTIONAL. Note that this keyword is different from the ISL OPTIONAL type constructor. The use of this keyword in ISL is deprecated in favor of the OPTIONAL type constructor.

  This also means that the ILU on-the-wire mapping for objects has been changed (slightly) to allow for NIL object references. Applications that do not use NIL objects will not encounter this change.
- The usage of the SINGLETON keyword on object types has changed. It now takes a string argument which defines the particular ‘‘pinfo’’ and ‘‘tinfo’’ to be used with the object type.
- Network GC now works.
- Numerous bugs have been fixed.

17.8 Bug Reporting and Comments

17.8.1 Known Bugs and Gotchas

KNOWN BUGS:

Release 2.0alpha10:

- *Java, C++* Network GC doesn't work with Java yet, and has never worked with our old C++ support.
- *kernel* UDP support is still broken.
- *C++* The current C++ support is old and buggy. There are known leaks in the generated stubs, and in the runtime itself. It is being completely re-worked for version 2.0 of ILU. We do not recommend serious use of the current C++ support, but it is included for use in testing other parts of the system.
- *Security* Still not released yet! We're updating our GSS implementation to the latest version of the spec, so we decided to hold it back for this release.
- *configuration* Since our Makefiles are constructed via imake from Imakefiles, which involves running the C preprocessor, watch out for use of predefined C preprocessor symbols in pathnames! Common boobytraps include names of processors, vendors, and operating systems (e.g., `"sparc"`, `"sun"`, `"hpux"`), which are used (as isolated tokens according to C rules) in some folks' conventions for naming directories. If you're lucky, you can solve these problems with quoting. A more heavy-duty approach is to #undef the offending macros at the start of imake/ilu.defs.new, and re#define them at the end.

- *ISL* "TYPE X = OBJECT SUPERTYPES Y ...; TYPE Y = OBJECT SUPERTYPES X ..." crashes the parser (and thus islscan and all the stubbers). Don't do this!
- *MOP* If ilu_AddRegisterersToDefault is used, callers of ilu_Register{Input,Output}Source must be prepared for false callbacks.

Release 2.0alpha9:

- *OMG IDL* The Java, Lisp, and C++ support in this release is fairly flakey, and in particular will not work with interfaces written in OMG IDL.
- *Java, C++* Network GC doesn't work with Java yet, and has never worked with our old C++ support.
- *kernel* UDP support is still broken.
- *[C++]* The current C++ support is old and buggy. There are known leaks in the generated stubs, and in the runtime itself. It is being completely re-worked for version 2.0 of ILU. We do not recommend serious use of the current C++ support, but it is included for use in testing other parts of the system.
- *[Security]* Still not released yet! We're updating our GSS implementation to the latest version of the spec, so we decided to hold it back for this release.
- *[configuration]* Since our Makefiles are constructed via imake from Imakefiles, which involves running the C preprocessor, watch out for use of predefined C preprocessor symbols in pathnames! Common boobytraps include names of processors, vendors, and operating systems (e.g., "sparc", "sun", "hpux"), which are used (as isolated tokens according to C rules) in some folks' conventions for naming directories. If you're lucky, you can solve these problems with quoting. A more heavy-duty approach is to #undef the offending macros at the start of imake/ilu.defs.new, and re#define them at the end.
- *[ISL]* "TYPE X = OBJECT SUPERTYPES Y ...; TYPE Y = OBJECT SUPERTYPES X ..." crashes the parser (and thus islscan and all the stubbers). Don't do this!
- *[MOP]* If ilu_AddRegisterersToDefault is used, callers of ilu_Register{Input,Output}Source must be prepared for false callbacks.

17.8.2 Reporting Bugs

Report bugs (nah! -- couldn't be!) to the Internet address `ilu-bugs.parc@xerox.com`, or to the XNS address `ILU-bugs:PARC:Xerox`. Bug reports are more helpful with some information about the activity; *please* read Section 16.7 [Debugging ILU Programs], page 236, for more information on how to look at problems. General comments and suggestions can be sent to either `ILU@parc.xerox.com` or `ILU-bugs`.

18 Multiple Languages In One Address Space

With ILU version 2.0 or later, modules implemented in different programming languages can be mixed in the same address space, with ILU doing automatic translation between data representations.[1] There are a number of things to consider when doing this; this section discusses some of them.

18.1 Dueling Runtimes

Some languages simply cannot be mixed in the same address space because their runtimes will conflict. ILU offers no solutions to this problem. Typical examples are two languages like Franz Allegro Common Lisp and DEC SRC Modula-3. They each implement a user-level threads package, and their implementations of threads probably cannot co-exist.

A possible solution to this problem, called the POSIX Portable Common Runtime (PPCR), is available from Xerox PARC, as `ftp://ftp.parc.xerox.com/pub/ppcr/`. It containsa basic runtime which can be used as the platform for a particular language implementation's runtime. Languages which use PPCR will have a lower chance of having conflicting runtimes.

18.2 Module Initialization

Module initialization really consists of two operations: interface initialization and object instantiation. The first operation initializes all the interfaces used or exported by the module; the second creates one or more true instances of objects to be used by other modules.

Generally, each ILU interface must be initialized. The process of doing this initialization varies from programming language to programming language. In ANSI C, ILU requires explicit calls to *interface*`__Initialize()` for interfaces being used, or *interface*`__InitializeServer()` for interfaces being exported. In C++, interface initialization is performed automatically, but at some indeterminate time before the first symbol from that interface is referenced from outside the interface. In Python or Common Lisp, interface initialization is performed automatically by the language at the time the module describing the interface is ''loaded'' into the address space.

In addition to initializing all interfaces being used or exported, a module must create one or more true object instances, to allow other modules to access it. Again, the specific way of doing this varies from programming language to programming language. Once the true instance has been created, it can be *exported* from the module by either publishing it, via the ILU simple binding system, or taking its string binding handle, and passing that outside the module for other modules to use.

[1] Currently, it does this by marshalling one language's representation into a data buffer, as if to transmit it to another address space, then hands this buffer to the other language's ILU support, which unmarshalls it into representations appropriate for that language. We plan to optimize this process in the future.

When multiple languages are used in the same address space, each must be initialized according to the standards used for that programming language. This can be tricky when using both statically compiled and dynamically compiled languages together. Consider the case where Python and ANSI C are linked together. This use of Python may be as an extension language to a program written in C. In this case, the C code must do all initialization of modules written in C before calling into any Python module which might reference them. Similarly, Python initialization (`import`) of modules must occur before the C code can use them. See '*ILUSRC*`/src/examples/multlang/`' for an example of a situation of this sort.

In the other case, C true modules which are to be used from a Python program in the same address space must somehow be first loaded into that address space, then initialized. The loading is done by turning the C module into a Python *extension module*, and either linking it into the `python` image, or creating a dynamically loadable module from it. The initialization is done by then calling `import` on that module from within the Python interpreter. The extension module's initialization routine initializes all of its interfaces, creates one or more true objects, and exports them. After the `import` statement returns, the objects are available for finding (see next section) from within Python.

18.3 Finding Objects

Object instances may be located by calls on the variations of `LookupObject` and `ObjectOfSBH` that exist in the various language runtimes. `LookupObject` is implemented so that it first looks to see if the true object for the specified object ID is available in the local address space. If so, it returns a version of that object. Only if the object is not locally available does it perform external lookups in an attempt to locate the object. Note that for an object to be found via `LookupObject`, the true instance must first have been published via the implementation language ILU runtime's variant of `PublishObject`. If you do not want your objects published outside your address space, you should use `ObjectOfSBH` to find them.

19 Using Imake with ILU

ILU uses the `imake` system from the X Window System distribution. `imake` provides a parameterized way of constructing '`Makefile`'s automatically from '`Imakefile`'s. The '`Imakefile`'s contain macros which are expanded to regular '`Makefile`' rules when the program `imake` is run.

19.1 Creating '`Makefile`'s from '`Imakefile`'s

The program `ilumkmf` is supplied with the ILU system. When run, it will use the '`Imakefile`' in your current working directory as input, and produce the corresponding '`Makefile`', again in the current working directory:

```
% cd myilu
% ls
Imakefile foo.isl fooProg.cc
% ilumkmf
% ls
Imakefile Makefile foo.isl fooProg.cc
%
```

19.2 ANSI C Usage

A typical '`Imakefile`' for an ANSI C ILU application would look like:

```
NormalObjectRule()  /* this rule defines the .c -> .o step */

InterfaceTarget(foo.isl)
ILUCTarget(foo.h foo-surrogate.c foo-common.c foo-true.c, foo.isl)

DepObjectTarget(programComponent1.o, foo.h somethingElse.h)
ObjectTarget(programComponent2.o)

CProgramTarget(program, programComponent1.o programComponent2.o foo-surrogate.o
foo-common.o,,)
```

19.2.1 ANSI C ILU `imake` Macros

The variable `LOCAL_INCLUDES` is a list of include file locations to be included when compiling.

The variable `ANSI_C_COMMAND` defines the particular command invoked for compiling ANSI C on your system. If you wish to use a different ANSI C compiler, override the default command by redefining this value in your '`Imakefile`'. Note that it may also be necessary to build a version of the ILU ANSI C library, '*ILUHOME*`/lib/libilu-c.a`', to use with this compiler.

`NormalObjectRule()` defines a number of suffix rules, in particular the one to go from '`.c`' files to '`.o`' files in your environment.

`InterfaceTarget(`*ISL-file*`)` defines a number of rules about the '`.isl`' file *ISL-file*. You should have one of these in your '`Imakefile`' for every interface you use.

`ILUCTarget(`*generated-files*`,` *ISL-file*`)` defines which ANSI C files are generated from the '`isl`' file and may therefore by re-generated at will, and when the '`.isl`' file changes. Generally, for an interface called `foo`, the generated files will be '`foo-surrogate.c`', '`foo-true.c`', '`foo-common.c`', and '`foo.h`'.

`ObjectTarget(`*object-file*`)` simply states that the specified *object-file* should be built.

`DepObjectTarget(`*object-file*`,` *dependencies*`)` says that the specified *object-file* should be built, and that it depends on the files specified in *dependencies*, which is a list of file names separated by spaces. Whenever something in the *dependencies* list changes, the *object-file* will be re-built.

`CProgramTarget(`*program-name*`,` *objects*`,` *dep-libraries*`,` *non-dep-libraries*`)` defines a program called *program-name* that is dependent on the object files defined in *objects*, and the libraries specified in *dep-libraries*, so that it will be re-built if anything changes in those two groups. It will also be linked with libraries specified in *non-dep-libraries*, but will not be re-built if they change. Note that the ILU ANSI C libraries are not automatically included by this command, but may be specified as part of the program by specifying them as part of either `dep-libraries` or `non-dep-libraries`.

`ILUCProgramTarget(`*program-name*`,` *objects*`,` *dep-libraries*`,` *non-dep-libraries*`)` defines a program called *program-name* that is dependent on the object files defined in *objects*, and the libraries specified in *dep-libraries*, and the normal ILU ANSI C libraries, so that it will be re-built if anything changes in those three groups, all of which will be linked into the program *program-name*. It will also be linked with libraries specified in *non-dep-libraries*, but will not be re-built if they change. This differs from `CProgramTarget` in that the ILU libraries are automatically included.

19.3 C++ Usage

A typical '`Imakefile`' for a C++ application and ILU would look like:

```
LOCALINCLUDES = -I$(ILUHOME)/include
ILULIBS = $(ILUHOME)/lib/libilu-c++.a $(ILUHOME)/lib/libilu.a

NormalObjectRule()  /* this rule defines the .cc -> .o step */

InterfaceTarget(foo.isl)
ILUCPlusPlusTarget(foo.H foo.cc foo-server-stubs.cc, foo.isl)

DepObjectTarget(programComponent1.o, foo.H somethingElse.H)
ObjectTarget(programComponent2.o)

CPlusPlusProgramTarget(program, programComponent1.o programComponent2.o foo.o,
$(ILULIBS),)
```

19.3.1 C++ ILU `imake` Macros

The variable `LOCAL_INCLUDES` is a list of include file locations to be included when compiling. `-I$(ILUHOME)/include` should always be on this list for compiling ILU applications.

The variable `CPLUSPLUS_COMMAND` defines the particular command invoked for compiling C++ on your system. If you wish to use a different C++, override the default command by redefining this value. Note that it will also be necessary to build a version of ILU C++ library, '*ILUHOME*`/lib/libilu-c++.a`', to use with this compiler.

`NormalObjectRule()` defines a number of suffix rules, in particular the one to go from '`.cc`' files to '`.o`' files in your environment.

`InterfaceTarget(`*ISL-file*`)` defines a number of rules about the '`.isl`' file *ISL-file*. You should have one of these in your '`Imakefile`' for every interface you use.

`ILUCPlusPlusTarget(`*generated-files*`,` *ISL-file*`)` defines which C++ files are generated from the '`isl`' file and may therefore by re-generated at will, and when the '`.isl`' file changes. Generally, for an interface called `foo`, the generated files will be '`foo.cc`', '`foo.H`', and '`foo-server-stubs.cc`'.

`ObjectTarget(`*object-file*`)` simply states that the specified *object-file* should be built.

`DepObjectTarget(`*object-file*`,` *dependencies*`)` says that the specified *object-file* should be built, and that it depends on the files specified in *dependencies*, which is a list of file names separated by spaces. Whenever something in the *dependencies* list changes, the *object-file* will be re-built.

`CPlusPlusProgramTarget(`*program-name*`,` *objects*`,` *dep-libraries*`,` *non-dep-libraries*`)` defines a program called *program-name* that is dependent on the object files defined in *objects*, and the libraries specified in *dep-libraries*, so that it will be re-built if anything changes in those two groups. It will also be linked with libraries specified in *non-dep-libraries*, but will not be re-built if they change. Note that the ILU ANSI C libraries are not automatically included by this command, but may be specified as part of the program by specifying them as part of either `dep-libraries` or `non-dep-libraries`.

`ILUCPlusPlusProgramTarget(`*program-name*`,` *objects*`,` *dep-libraries*`,` *non-dep-libraries*`)` defines a program called *program-name* that is dependent on the object files defined in *objects*, and the libraries specified in *dep-libraries*, and the normal ILU ANSI C libraries, so that it will be re-built if anything changes in those three groups, all of which will be linked into the program *program-name*. It will also be linked with libraries specified in *non-dep-libraries*, but will not be re-built if they change. This differs from `CProgramTarget` in that the ILU libraries are automatically included.

Appendix A How to Create ILU Support for a Programming Language

This chapter is addressed to the ILU developer who wants to add support for a new programming language to ILU.

Supporting a new programming language for ILU involves four tasks: (1) defining how ISL is mapped into that programming language; (2) designing how stubs work, and their interface to the *Language-Specific Runtime* (*LSR*) support; (3) writing the *stub generator* (*stubber*) for that language; and (4) writing the Language-Specific Runtime (LSR) support. Naturally the existing language supports provide interesting examples when defining a new one. The best existing supports are currently those for C and Python. Understanding of the other generic parts of this reference manual is assumed.

Adapting ILU to work on different operating system interfaces is an independent topic, that of *porting* (not addressed in this chapter).

A.1 Defining the Mapping

The problem here is to define how each of the concepts in ISL maps into a concept in the programming language. We prefer to map into the most natural presentation for the language at hand. For example, a `SEQUENCE OF SHORT CHARACTER` maps into a C `char *`, because `char *` is by far the most conventional way to represent a string in C, even though no tasteful person would make that choice if starting from scratch today.

When working on a language *L* for which the OMG has defined a CORBA mapping, we strive to satisfy the following equation:

```
for all acceptable IDL i: ISL-to-L(IDL-to-ISL(i)) = IDL-to-L(i)
```

where IDL-to-L is the mapping defined by CORBA, IDL-to-ISL is our standard way of translating CORBA interfaces into ILU interfaces, *acceptable IDL* is IDL that's acceptable to our IDL-to-ISL translation, and ISL-to-*L* is the mapping being defined [hmm, this equation should probably be a little more complicated, to take into account the fact that one IDL source can declare multiple "interfaces"]. If there is no ISL-to-*L* that satisfies the above equation, we also have to modify IDL-to-ISL (and thus all the other ILU language mappings too --- so we don't want to do this very often!).

When working on a language for which there is a defined IDL mapping, it's also necessary to have a mapping for the standard CORBA "psuedo-objects", and the methods of `CORBA::Object`. Even when working without interference from CORBA, it's worth considering the issues those features address. ILU does not (and thus your language mapping need not) attempt to provide the following parts of CORBA: Interface Repository, Implementation Repository, Dynamic Invocation Interface, Dynamic Server Interface, Basic Object Adaptor, Portable Object Adaptor (yet).

ILU also has some general (i.e., not explicitly appearing in any particular ISL file) features that CORBA doesn't and must be addressed in a language mapping; prominent among them are servers, and their attendant ports and object tables. You will need to define how servers are created and shut down; this should be able to follow the spirit of the kernel interface on this matter closely. Define how an application indicates which of its true objects are in which of its true servers. You may want to provide a "default server" for use in simple programs that don't want to explicitly manage multiple servers. ILU also involves explicit management of ports, to give applications control over selection of RPC and transport protocols used; define how ports are created and destroyed. ILU supports (but not as completely as we'd like) the possibility of multiple ports on a server, with inclusion of information about all those ports in an object reference (string binding handle, "SBH", in ILU parlance; see Section 1.3.1.6 [String Binding Handle], page 7). Define how an application supplies an object table to a server.

ILU's has its own security story. This story only concerns how to provide authentication, message integrity, and message secrecy --- or a selected subset of them. The paradigm is that a security filter can appear in the protocol stack; higher layers need only concern themselves with getting the necessary parameters to and from that security filter. The necessary parameters are bundled into an `ilu_Passport`, which is a collection of `ilu_IdentityInfo`s. Each call may optionally have an `ilu_Passport` associated with it. You design the language-specific appearance of passports and identities, and a calling convention that incorporates implicit optional parameters.

There are two more implicit optional parameters, and they are of interest only on the client side of a call: a serialization guarantee instance (`ilu_Serializer`) and an pipeline (`ilu_Pipeline`). You design the language-specific appearance of serialization guarantee instances and pipelines, and further use of the calling convention that incorporates implicit optional parameters.

A language mapping should preserve the type structure of the ISL. In particular, the subtyping relationships should be preserved. That is, if ISL type *T1* is, according to ISL semantics, a subtype of ISL type *T2*, then *LT1* (the language-*L* mapping of *T1*) should be, according to language *L*'s semantics, a subtype of *LT2*.

A.1.1 Object References

The biggest part of this definitional challenge usually concerns objects. Many programming languages do not have an object system as rich as ISL's; some have no object system at all. If not already present, an object system of sufficient richness must be invented and embedded in the programming language. C is one example of this: it has no object system at all, and so the OMG invented an embedded one for use with CORBA. Modula-3 is an even thornier example: it has an object system, but with only single inheritance; for an ILU mapping, a multiple-inheritance object system has to be embedded in Modula-3 --- in the least annoying way possible (and exactly what is least annoying is unfortunately debatable).

A.1.2 Storage Management

Another major part of the defintion to be made concerns storage management (and for objects, management of other, hidden, resources). Prominent areas where this issue shows up are in management of objects, and of arguments, results, and exception parameters of method invocations. Servers and the call-specific meta-objects (passports, identities, pipelines, and serialization guarantee instances) also need management. In the simplest case, you map into a language that is defined to include a garbage collector, and do not aspire to enable the programmer to free resources before the collector eventually does. However, even when a garbage collector is available, some application programmers may demand to be able to release the resources hidden behind an object (e.g., private state, file handles, ILU kernel objects) when the application detects that it's done with the object, which (for most collectors) can be arbitrarily long before the collector does. When a garbage collector is not available, or when explicit management by the application is desired, a reference-counting scheme --- or something even more painful --- is often employed. Some languages are not defined to include a garbage collector, but add-on collectors are available; for such a language, the most flexible design would cope with operation both with and without a collector.

A.1.3 Control Flow

Another issue that must be addressed in defining a language mapping (and even more so in the following steps) is control structure. ILU is designed to be usable in both single-threaded and multie-threaded environments. Some languages are defined to be one or the other. Other languages (such as C and C++) are single-threaded `"by default"`, but can be used to write code for use in a multi-threaded environment. Your mapping should support the same choices in this area as the language does.

In a single-threaded environment, many packages use a `"main loop"` that does `"event dispatching"` as their top-level control structure. ILU is such a package, on the server side. Unfortunately, some programs must integrate two or more such packages --- and are thus presented with a problem: there can't be two or more *top-level* control structures. Solving this problem requires cooperation from the main-loop-using packages involved. The solution involves picking one package's one main loop (or inventing a completely independent loop) to be the top-level control, and making the other packages `"adopt"` the chosen main loop for their own use. ILU is designed to be able to play either role: that of `"donating"` or `"adopting"` main loop services; your language mapping (for a single-threaded language) should expose the support for both roles.

In a multi-threaded environment, there are issues of synchronization between threads. At the language mapping level, these are mostly independent of what you must define. Just follow this simple rule: ILU objects and their methods allow just as much concurrency as the other objects and methods in the language. Of course, the implementations of ILU objects will involve synchronization issues, but that's a subject for later sections. The one place where synchronization typically shows up in a mapping defintion is in object tables, whose methods are called under certain mutual exclusion conditions.

See Chapter 12 [Threads and Event Loops], page 191 and the `"Locking"` section of '`iluxport.h`' for more details on mutexes and locking comments.

Your language mapping must include locking comments in the defined interfaces (e.g., the definition of an object table, and interfaces emitted by the stubber). One or the other of the above views (invariant-oriented or object locking) must be complete; both would be nice.

A.1.4 Any and TypeCode

ILU `PICKLE` support provides dynamically-typed value functionality in a strongly-typed object interface system. This is typically useful for intermediary modules, such as name services, which hold onto arbitrary values associated with some other value, but never need to use those values except to send them to some other module. `PICKLE` itself is treated as a basic type in ILU ISL, much like `BOOLEAN` or `REAL`.

The `PICKLE` support in ILU is modelled after the *pickle* concept in several other programming systems, such as Modula-3 and Python. The idea is that a value is represented as a tuple `(`<*type-id*>`,` <*marshalled-value*>`)`, where the <*type-id*> identifies the value's type, and the <*marshalled-value*> is a representation of the value in some standard form. In ILU, we use the standard ILU type id (or CORBA repository id) for the <*type-id*>, and the standard representation form is a simple big-endian, non-aligned form. The tuple itself is bundled up into a sequence of bytes, for more convenient handling in the ILU kernel and with the various wire protocols.

The language runtime should provide general compatibility with CORBA's notion of `any`, which is another dynamically typed value type. This basically means that there should be two kinds of objects added to those supported by the language runtime: *Any* and *Typecode*. The *Any* object is used as the language-specific representation of an ILU pickle; the *Typecode* object is used to represent an ISL type.

The *Any* object LSR support is quite simple. It should be implemented as an opaque object with accessors. This allows the value contained in the *Any* to be kept in the pickle form until it is asked for, which makes the operation of ILU pickles particularly efficient in services such as name services or event services, in which the values are never unpickled. There should be a constructor function for *Any* which allows an instance to be constructed from a *Typecode* and a language-specific value. There are minimally two methods on the *Any* object, one to retrieve the *Typecode* of the contained value, and one to retrieve the contained value itself in language-specific form. It is allowable to refuse to return a language-specific value if the *Any* contains a pickled value the type ID of which is unknown to the language runtime.

LSR *Typecode* support is also simple. There should be a method to retrieve the type ID of the *Typecode*, and another to compare two *Typecode* values for equality. *Typecode* values are typically generated by the stubber, one for each type defined in an interface's ISL. The language runtime must also provide *Typecode* values for the ILU basic types, such as `REAL` or `BYTE`. To aid in that, the kernel exports a set of character arrays containing the type ID's for those basic types; see the file '*ILUSRC*`/runtime/kernel/ilutypes.h`' for the names of these arrays.

A.2 Designing Runtime Operation

This task mainly involves designing what the generated stubs look like, prominently including: the data structures that represent ILU objects, the way the storage management issues are implemented, and the sequence of LSR calls involved in making and serving ILU method calls. For the first two, see Section A.4.3.5 [Object Management], page 285 for a discussion of the options. We have found the last one to be facilitated by writing some example stubs by hand, and writing down a draft LSR interface. A pre-requisite to doing these tasks is understanding how the ILU runtime kernel is designed to support LSRs (which is discussed in the section on writing the LSR).

Another issue to address in runtime design is how the control structure options defined in the language mapping are implemented, and the impact on the rest of the runtime design.

It is important to understand the major pieces of a running ILU program. At the base, there is some operating system interface, and libc (the standard ANSI C runtime library). On top of the OS and libc is the ILU runtime kernel, a common piece of runtime support. It is written in ANSI C, used in each of ILU's language runtime supports, and does as much of that job as possible. The remainder of the job is done by the LSR, which is just a thin (as thin as possible, but no more) veneer over the kernel. When multiple languages can co-exist in one runtime environment, their ILU LSRs should also be able to co-exist; the kernel is prepared to deal with multiple LSRs.

A.3 Writing the Stub Generator

Each stub generator is an independent program, and can (in principle) be structured in any way, and written in any language. Most of the existing stubbers are written in ANSI C, and make use of a common front-end (found in '`stubbers/parser/`'). This parser makes use of a BISON grammer (in '`ilu.bison`', translated to '`iluparse.c`'); this file also includes some procedures of common utility to back-ends. The interface to the parse tree and common utilities is '`iluptype.h`'.

Much could be written about '`iluptype.h`', but right now I'll give only a few clues. First, it could profit from significant revision; sorry about that.

The type `Procedure` refers to a method of an object type. The type `Class` refers to an object type (ILU is full of places where "class" is used instead of "object type" --- because the word "class" was erroneously used for "object type" early in the project).

The type `Type` is confusing in many ways. The type `Type` refers to an occurence of a type reference in the parse tree --- not the underlying, analyzed type. Analyzed type information is found in a `TypeDescription`, which is hung off the `Type` appearing in the definition statement (`TYPE` *name* `= ...;`) that introduces that type information. For other `Types`, the `supertype` member (grossly misnamed) points toward another `Type` closer to the one that holds the `TypeDescription`. Type `Type` has two members that identify interfaces: one (`Interface interface`) identifies the interface in which the type reference appears; the other (`string importInterfaceName`) identifies the interface into which the reference is referring (i.e., the `Ifc` part of a type reference that looks like `Ifc.Foo`).

A good example might be the Python stubber, which is highly organized and clean.

A.4 Writing the LSR

This is about writing the veneer between (a) the kernel and (b) the stubs and application. Some of this topic is addressed in the refman, and some in the kernel's interface ('`runtime/kernel/iluxport.h`').

A.4.1 Control Structure Options

By default, the kernel operates single-threaded, using ILU's default main loop. If your language is multi-threaded, your LSR will need to make calls at startup time to switch the kernel into multi-threaded operation. If your language is single-threaded, your adopt-a-foreign-main-loop operation will need to call into the kernel to make it adopt a new main loop (you'll have to provide a "glue" object that translates between the ANSI C declaration of a main loop meta-object in the kernel interface and the language *L* declaration in your runtime interface). See Chapter 12 [Threads and Event Loops], page 191 for more details.

To switch the kernel to multi-threaded operation, four procedure calls must be made early in the initialization sequence. The procedures to be called are: `ilu_SetWaitTech`, `ilu_SetMainLoop`, `ilu_SetLockTech`, and `ilu_NewConnectionGetterForked`. See '`iluxport.h`' for details, and the Java and Common Lisp LSRs (found in '*ILUSRC*`/runtime/java/`' and '*ILUSRC*`/runtime/lisp/`') for usage examples.

Instead of the first three procedures, an LSR may call `ilu_InitializeOSThreading` to use predefined metaobjects constructed from the POSIX, Solaris 2, or Win32 threads API. This option is only available if running on a system that exports one of the above threads APIs, and only if `--enable-os-threads` was selected during the configuration step of installing ILU (see Section 17.2.4.1 [Configuration Options], page 242).

See the "Concurrent I/O routines" section of '`iluxport.h`' for more details on this issue: how the kernel donates and adopts main loop services and FD waiting services.

Remember to rigorously document the use of mutexes in your internal interfaces and code. Verify it meshes properly with the locking comments in the kernel's interface.

A.4.2 The Main Sequence - How Calls are Handled on the Client and Server Sides

This should be well documented in the "Client side routines" and "Server side" sections of '`iluxport.h`'.

A.4.3 Object Management

For each language *L*, kernel object *ko*, and corresponding language-*L* *lso*, the object's server includes this invariant: either (a) *ko* and *lso* both exist and point to each other, or (b) neither points to the other (and either or both may have been freed). This organizes the relationship between the two objects, which otherwise could be created and destroyed rather independently (with ensuing confusion in any code that must relate the two). More details are in the "Deleting Objects" section of '`iluxport.h`'.

A.4.3.1 Discussion

An ILU object is represented in a program instance (aka running program, UNIX process) by a kernel object (KO) and, for every programming language X in which that object is being manipualted, an

Application-level Language-Specific Object (App LSO for language X). The X App LSO is an object in the object system of the ILU mapping to X. In fact, there may be multiple X App LSOs corresponding to one ILU object --- when CORBA::Object::duplicate isn't a no-op. Between an App LSO and its KO there may be a number of auxiliary objects that implement the defined relationship between the two main objects; like the KO, these auxiliary objects are not exposed to the application.

Both the KO and the X App LSO (or one of its associated auxiliary objects) have slots that hold pointers to the other (or NIL). The KO's slot that holds a pointer to the X LSO is accessed via getter and setter procedures in the kernel interface, `ilu_GetLSO` and `ilu_SetLSO` (these are the shorthands I'll use here for whatever their real names in iluxport.h are). The X LSO's slot that holds a pointer to the KO is the X LSR's responsibility to declare.

The ILU runtime for language X can be designed such that X App LSOs are managed by a garbage collector, or such that they are explicitly managed by the application --- or even to leave that choice up to the application. The KOs are explicitly managed by the kernel and LSRs. A KO will persist as long as it has any associated LSOs -- or any of certain other reasons (which are private to the kernel) to persist. One of those other reasons is being of a collectible MST and having recently had live surrogates. The LSR's role in the explicit management of a KO is captured in its calls on `ilu_SetLSO`: while the most recent call passed a non-NIL value for 'lso', the LSR is deemed to have some App LSOs associated with the KO. Because the other reasons for a KO's continued existence are private to the kernel, an LSR can rely on the KO still existing only when its most recent call on `ilu_SetLSO` passed a non-NIL 'lso'. For this reason we have the following invariant: either (a) the KO and X App LSO point to each other, or (b) both pointers are set to NIL (and each object is being used independently or has been destroyed). This invariant is part of the mutex invariant for the object's server's mutex. In other words, the server's mutex must be held when reading or writing either of these pointers.

KO and App LSO can exist and be used independently of one another --- for uses that don't logically require both. For example, an application can create App LSOs and use them only locally --- and as long as there's no attempt to marshal or unmarshal the LSO (or do any other ILU-specific thing with it), there will be no KO or auxiliary objects involved. And of course you would expect that in a multi-language runtime environment, the existence of a KO and X App LSO does not require the existence of a Y App LSO. The kernel can serve and make built-in calls using only the KO. In special times during the startup and shutdown sequences of objects, you might also expect an KO to exist while there is no corresponding App LSO.

In some cases, a dissociated KO and X App LSO can be found and linked together by the ILU X runtime; sometimes this can even involve creating one of those two when starting from the other. The cases depend on whether we're talking about a true or surrogate object, and, in the true object case, whether the object's server has an object table.

The ILU X runtime should be able to create from scratch a surrogate App LSO for any given ILU object type that has been registered (by a stub) with that runtime. In general we expect the aforementioned registration step to supply a procedure for instantiating a new App LSO of the appropriate language type;

in a language with sufficiently powerful reflective capabilities, the runtime can do the instantiation without calling type-specific code.

The ILU X runtime is not expected to be able to create a true App LSO; only the application is expected to be able to do that. The application can always create App LSOs on its own initiative. Additionally, if the application supplies an ‘object table’ to the server, the ILU kernel and X runtime can call on this object table to create App LSOs when needed. Remember that the object table that the application supplies will create true App LSOs, while the object table of the kernel server must return KOs (that are true for the server’s language).

The kernel is able to create a KO given an SBH, or (equivalently) an instance handle and a server. At the kernel interface, this ability is bundled into procedures that either find an existing KO or create a new one. For creating a KO that is true for some language, there are two cases: (1) the caller wants the kernel to consult the server’s object table (if any), and (2) not. Case (2) is useful when the ILU X runtime is already holding the App LSO, and just wants the corresponding KO; case (1) is useful when starting from just the SBH. Case (1) is `ilu_ObjectOfSBH`; case (2) is `ilu_FindOrCreateTrueObject`; `ilu_FindOrCreateSurrogate` is another variant, for which object tables are irrelevant.

An App LSO should be able to identify the SBH, or the instance handle and server, of the ILU object in question. There are any number of ways of formulating this. It is most interesting for true objects, because this is a question of how this responsibility is put upon the application. In the C mapping, the procedure for creating a true App LSO takes instance handle and server as arguments. In the Modula-3 mapping (now defunct), the App LSO had to implement a "get-server" method, and the server’s M3 object table has an additional "instance-handle-from-object" method (a default implementation being used for servers that lack object tables).

There must be some way to determine when an application is done with the App LSO(s) of a given ILU object. In some languages, a garbage collector is available to do this. To be useful, it must be possible to register a ‘finalization’ procedure that can be run after the collector determines an object is unreachable. If this is not possible, garbage collection cannot be used to manage App LSOs. Furthermore, there must be a ‘WeakRef’ facility. A WeakRef is a thing that holds a disguised pointer (i.e., one that doesn’t count as keeping its target ‘reachable’) to some object; there is a ‘reveal’ operation on the WeakRef that returns a normal pointer to the obscurely-referenced object. If and when the collector decides the obscurely-referenced object is unreachable, it atomically changes the WeakRef such that future invocations of ‘reveal’ will return NIL.

In languages without a useful garbage collector, some other method must be used to determine when an App LSO is no longer useful to the application. The CORBA-mandated solution is the most useful one. It can be understood as reference counting. The App LSO (or one of its auxiliary objects) holds a reference count. Every unmarshalling of the object increments the count. CORBA::Object::duplicate is used to produce an independently managed reference from some existing reference; this procedure increments the reference count and returns the given reference. CORBA::Object::release is used to declare the application done with (a copy of) the reference; it decrements the count. When the reference count reaches 0, all parts of the application are done with the App LSO.

A.4.3.2 Life Cycle

Following is a summary of the life-cycle of a KO and App LSO (and the linkage between them).

One startup path begins with the application creating a true App LSO. This may be on the application's initiative, or as the result of an object table invocation. Creation of the App LSO does not by itself require creation of the KO or any linkage between them; the application can now do arbitrary non-ILU things with the App LSO. At some future point the X LSR may need to get the corresponding KO. If the KO and App LSO are already extant and cross-linked, the pointer from App LSO to KO reveals this, and the LSR is done. Otherwise, the kernel is consulted, via `ilu_FindOrCreateTrueObject`; this procedure returns a KO, whether it existed before or not. The LSR cross-links the KO and App LSO.

Let us digress a moment to think carefully about object tables. The API for ILU's X mapping includes a declaration of the type of object tables; such an object table has an 'instance-handle-to-object' method that returns an X App LSO. The implementation of this method is nearly arbitrary application code (the restriction is that it must run inside the server's mutex). The kernel interface uses a different type of object table, one whose 'instance-handle-to-object' method returns a KO. This object table is implemented by the LSR, in terms of the application-supplied object table. The instance-handle-to-KO method first calls the instance-handle-to-App-LSO method, and then continues as outlined in the latter half of the previous paragraph, starting from where the LSR is holding an App LSO and wanting a KO.

Another startup path is unmarshalling an object. The LSR first calls a kernel routine (`ilu_InputObjectID`) that returns a KO. If that KO is already cross-linked with an X App LSO, the LSR is done. Otherwise, two different courses of action are pursued, one if the object is true for X, the other if not. For the surrogate case, the LSR instantiates a new App LSO if possible, and cross-links it with the KO; if instantiation is not possible, the unmarshalling fails. For the true case, `ilu_InputObjectID` will have already invoked the kernel interface version of the server's object table (if any were present and needed) --- and this would have cross-linked the KO and App LSO; the present lack of cross-links means the unmarshalling should simply fail at this point.

Translating an SBH into an X App LSO proceeds much like unmarshalling; the difference is that the LSR starts by calling `ilu_ObjectOfSBH` instead of `ilu_InputObjectID`.

Having considered the ways that KO and App LSO can be created and cross-linked, we now turn our attention to ways they can be dissociated (un-cross-linked) and destroyed.

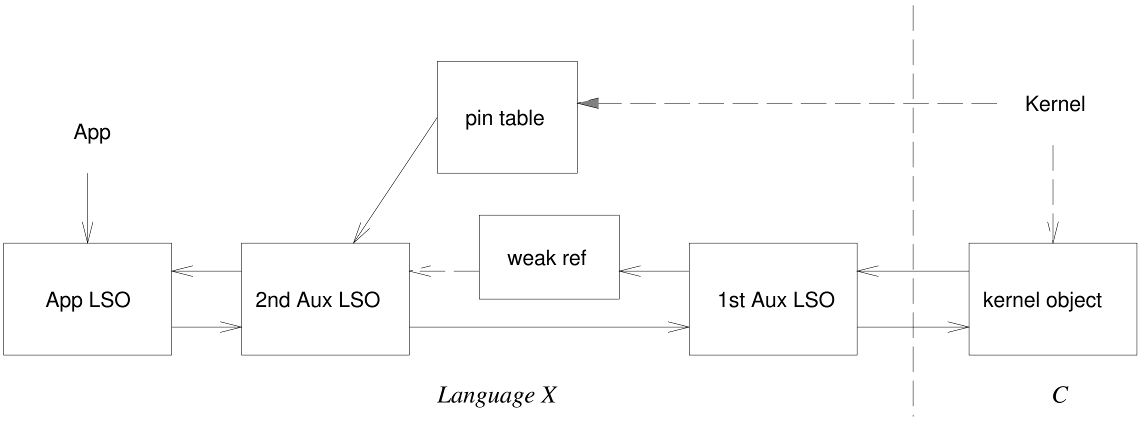
The LSR may supply the application with an operation to explicitly dissociate the KO and App LSO. This leaves the KO to be destroyed by the kernel when the kernel has no other reason to keep the KO around, and the App LSO to be destroyed by the X language runtime (perhaps augmented by the X LSR with a reference counting scheme) when the application has no more need for it. However, we should also consider what happens if, at any time after this dissociation, the LSR finds itself again wanting to cross the bridge between the two objects. Starting from the App LSO, this is easy, because the kernel will gladly return the still-existing KO or create a new one (whichever is appropriate). Starting from the KO, consider whether the App LSO is true or surrogate. In the surrogate case, the LSR could easily instantiate a new App LSO. But the old one might still exist, and we would then have a violation of ILU's 'EQ-ness preservation' property. Perhaps an application that explicitly dissociates KO and App LSO deserves what it gets in this case. If not, the LSR could keep its own instance-handle-to-App-LSO table (the LSR can arrange to know when the LSR is destroyed, and remove this table entry at that time), enabling the LSR to re-link the KO with the same old LSO if it still exists (and a new one otherwise). In the true case, the KO can be re-linked to an App LSO in the usual ways (by object table invocation or explicit true App LSO creation) and no others.

The other shutdown scenario starts with the LSR determining that the App LSO is no longer interesting to the application and no longer 'very interesting' to the kernel (as revealed by the `ilu_ObjectNoter` and/or `ilu_VeryInterested`). The LSR dissociates the KO and App LSO. This step will destroy the KO --- if it's still not interesting to the kernel. Again, the issue of possible later re-linking the two comes up, and the analysis of the previous paragraph applies; it shows that The Right Thing happens --- at least, as long as the App LSO cannot change from being unreachable back to reachable.

The remainder of this document gives some worked examples.

A.4.3.3 Automatically Managed Objects

Here's a basic scheme for garbage-collected runtimes:



This is a very generic scheme, designed to minimize demands on the language runtime and application. Certain of the above objects can be coalesced when more demands can be made (we'll get explicit about this later). The pointers to X objects are visible to the garbage collector (and thus might not be simple pointers). The pointers to C objects are C pointers; we assume that either (a) the garbage collector either doesn't manage the C heap, or (b) it manages the C heap and can follow C pointers.

In this scheme ILU's language X runtime establishes finalization on the 2nd Auxiliary LSO; the 2A LSO is distinct from the App LSO exactly so that the Application can put its own finalization on the App LSO.

The 1A LSO could be coalesced with the WeakRef if the language doesn't require them to be distinct objects.

The 1A LSO could be merged into the 2A LSO (leaving the KO to point to the WeakRef, which in turn points obscurely to the 2A LSO) if either (a) an object can be made reachable again in its finalization procedure and then re-subjected to finalization (the 2A LSO will want two different finalizations, but the App LSO just one to run once, of course), or (b) fairly heaveyweight stuff can be done within a finalization procedure.

The pin table holds a pointer to the 2A LSO exactly when the kernel is 'very interested' in the object; the LSR registers an `ilu_ObjectNoter` with the kernel to maintain this relationship.

Here's how the LSR gets an App LSO from an SBH. First, it calls the kernel's `ilu_ObjectOfSBH` procedure. This may, under the covers, call the server's object table. If `ilu_ObjectOfSBH` returns NIL, failure; otherwise, we're Inside(server, static_type). Next call `ilu_GetLanguageSpecificObject`; if this returns NIL, we: create a 1A LSO, create a Weak Ref that holds NIL, and call `ilu_SetLSO`. If the Weak Ref holds NIL, we consider whether the LSR can create the App LSO; this would normally be trivially true for a surrogate, and an object table consultation for a true object (note that this makes the external behavior independent of when the auxiliary process below gets around to `ilu_SetLSO`). If the WeakRef holds NIL and the LSR can create the App LSO, the LSR creates the App LSO and 2A LSO and sets the Weak Ref to (disguisedly) point at the 2A LSO; if we can't create the App LSO, failure. We return the App LSO to which the 2A LSO points, after exiting the appropriate mutexes.

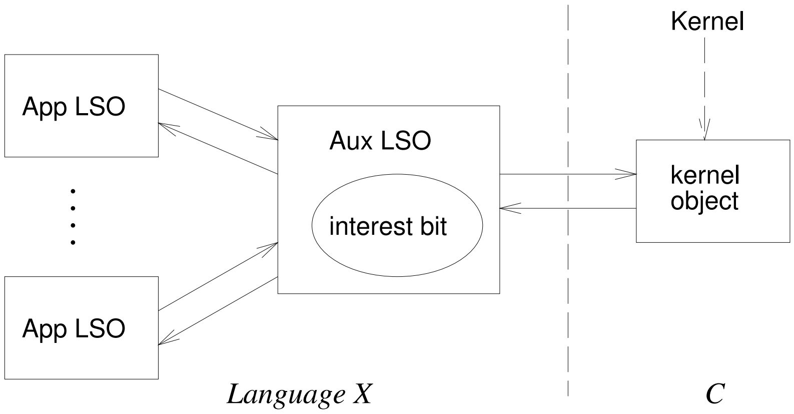
When the App LSO eventually becomes unreachable (from both the App and the pin table), the garbage collector should eventually schedule both the App LSO and the 2A LSO for finalization, setting the Weak Ref to NIL. If the X-Object-from-SBH procedure is executed after this point in time but before the finalization happens, the existing 1A LSO and Weak Ref will be traversed, to find the Weak Ref holding NIL, and a new 2A LSO and App LSO created if possible (as explained above). The finalization procedure puts the 1A LSO on a queue to be processed by an auxiliary thread or event handler. To avoid allocating in the finalization procedure, the queue is threaded through the 1A LSO. The auxiliary thread or event handler does the following for each 1A LSO it pulls from the queue. First, it establishes Inside(object's server's mutex, object's MST). Then it considers whether the auxiliary objects are the remnants of a forcible dissociation or the Weak Ref currently holds a NIL pointer. In either case, it is done; otherwise, it calls `ilu_SetLSO`, passing NIL as the `"lso"`. Finally, it disestablishes Inside(..).

An App LSO can be forcibly dissociated from the kernel object. This involves entering the object's server's mutex and: setting the four pointers to/from the KO and App LSO to NIL; leaving the three pointers among the 1A LSO, 2A LSO, and WeakRef intact; and removing the LSO from the pin table. An application may want to do this as part of a graceful shutdown procedure. This leaves the App LSO fully extant and constructed as far as X code can tell, except that it's not linked into the ILU runtime.

When relying on a garbage collector, CORBA::Object::duplicate is the identity function, and CORBA::Object::release is a no-op.

A.4.3.4 Explicitly Managed Objects

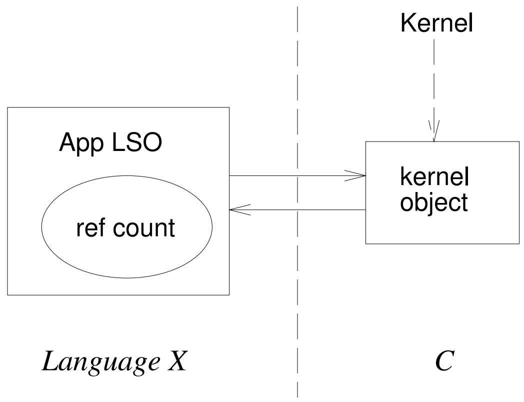
An alternative is to not rely on a garbage collector, and require the application programmer to use CORBA::Object::duplicate and CORBA::Object::release to explicitly manage objects. The most straightforward approach is this:



The 'interest bit' records whether the kernel is 'very interested' in this object. CORBA::Object::duplicate creates a new App LSO, as does each unmarshalling of the object. CORBA::Object::release deletes an App LSO --- both removing it from the set managed by the Aux LSO, and freeing it (what if true object?). When there are no App LSOs, and the interest bit is clear, the Aux LSO can be freed, and `ilu_SetLSO`(ko, NIL) called.

When multiple App LSOs are associated with a given abstract ILU object (i.e., kernel object), things get more interesting for true objects. We should not ask the application to implement cloning and consistency among clones. Instead, one of the App LSOs is true, and the others produced from it by CORBA::Object::duplicate or unmarshalling should be surrogates. ILU's language X stubber and runtime support should conspire to short-circuit calls made on a surrogate into calls on the true object, without involving the kernel. As far as the kernel is concerned, the Aux LSO is the true LSO. The short-circuiting could be done by structuring a client-side stub as follows: (1) am I a clone of the true object? (2) if so, call on the true object, otherwise (3) do the 'normal' thing and call through the kernel.

If there is no state that needs to be associated with individual App LSO copies, we can instead use just one copy, with a reference count. That would look like this:

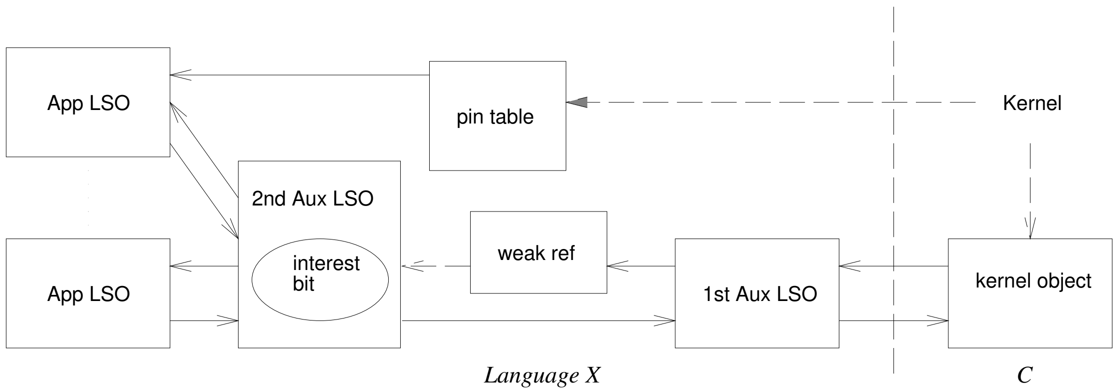


The kernel being ‘very interested’ in the object contributes 1 to the reference count. CORBA::Object:duplicate, and each unmarshalling, increments the reference count (and returns the same LSO), while CORBA::Object::release decrements it. When the reference count goes to 0, CORBA::Object::release breaks the cross-pointers between the KO and the App LSO and then frees the App LSO.

To forcibly dissociate the KO and App LSO, just break the pointers between them.

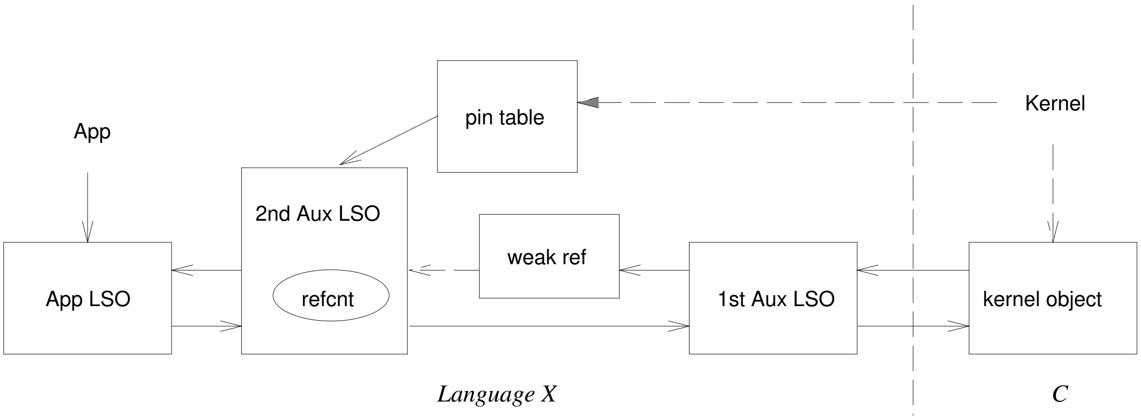
A.4.3.5 Hybrid Schemes

To give the App the choice, independently for each abstract ILU object (i.e, each kernel object), of either: (a) letting the GC reclaim all App LSOs or (b) explicitly managing all App LSOs:



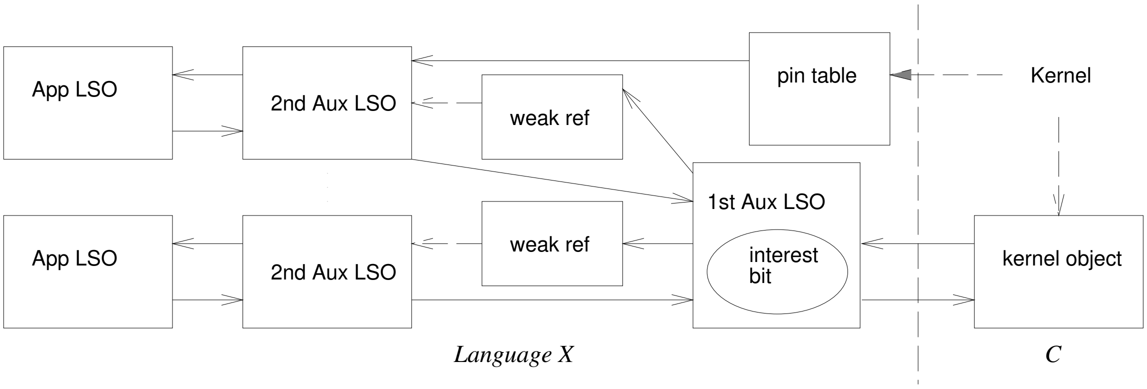
In this scheme we free the aux LSOs when either (a) the GC informs us that the 2A LSO is unreachable or (b) the last App LSO is deleted and the kernel is not ‘very interested’ in the object. CORBA::Object::release destroys an App LSO. The pin table points to the true App LSO, if any.

Again, if we don’t need distinct App LSOs, we can use one with reference counting:



This scheme requires the GC vs. explicit management choice to be made once per abstract ILU object.

To enable the choice independently at each App LSO:



Here the LSR establishes finalization on each 2A LSO. The finalization, like CORBA::Object::release, removes the relevant WeakRef from the 1A LSO's set of WeakRefs. When that set is empty, and the kernel is not 'very interested' in the object, the remaining LSO cruft can be freed, and `ilu_SetLSO`(ko, NIL) called. The pin table points at the true App LSO, if any.

A.4.4 Server Management

Like objects, servers are also managed. However, their management is typically easier, because they are not first-class objects (i.e., can't appear in network interfaces) and so are manipulated in only a few ways. You will define a *LSS* (*Language-Specific Server*, the language mapping of an `ilu_Server`).

Each `ilu_Server` has a collection of LSSes associated with it, in the same way that an `ilu_Object` has a collection of associated LSOs. `ilu_GetLSS` and `ilu_SetLSS` manipulate the pointer from the `ilu_Server` to its language-*L* LSS; the reverse pointer is your problem.

`ilu_Servers` are reference-counted, in a way similar to `ilu_Objects`. The server mutex invariant includes the assertion that, for each given language *L* an `ilu_Server` and it's *L* LSS point to each other, or neither does; each pointer from `ilu_Server` to LSS counts as a reference. Each locally-reified object in the server also counts as a reference.

The time when an `ilu_Server` is checked and possibly freed is when its mutex is exited. This should always be done by calling `ilu_ExitServerMutex` instead of fetching the mutex and applying the generic `ilu_ExitMutex` to it.

A.4.5 Call-Specific MetaObjects

`ilu_Pipeline`s and `ilu_Serializer`s are managed by a simple, explicit reference counting scheme: you get a reference count of 1 when you create one, and may later release that reference count. Each `ilu_Call` that is associated with one of these metaobjects also counts as a reference, if `ilu_StartCall` succeeded and `ilu_FinishCall` hasn't yet been called.

`ilu_Passport`s and `ilu_IdentityInfo`s are managed by an even simpler scheme. `ilu_DestroyPassport` frees a `ilu_Passport` and those of its `ilu_IdentityInfo`s that are marked as being owned by the `ilu_Passport`.

A.4.6 Errors

The kernel has a (partly deployed) standard way to return result codes and further details. It's called the "error system". '`runtime/kernel/iluerror.h`' describes the system in general terms, and '`iluerrs.h`' describes the particular types of errors currently possible. Most calls into the kernel take an error-signalling OUT parameter in the last position.

CORBA defines some standard system exceptions. An LSR must, when relaying an error from the kernel to an application, convert a kernel error into a CORBA standard system exception. To facilitate this, the kernel error system has been defined to include a collection of error types that correspond to CORBA's standard system exceptions.

Sadly, CORBA's standard collection of system exceptions doesn't correspond very directly to the full collection of all the things that can go wrong at the kernel interface. So there are some additional error types. Of course, when translating to a CORBA system exception, a standard one must be chosen. Also, there are some old error types left over from a previous version of the system; we are working on identifying which should be eliminated, and which should stay (perhaps with modification).

A.4.7 Internal Consistency Checks

The kernel includes internal consistency checks, of two styles. When such a check fails, one of a few possible courses of action are taken. An application can specify what to do. Your mapping should expose this ability. See the "Internal Consistency Checking" section of '`iluxport.h`' for details.

The kernel is allowed to check consistency only with respect to conditions that it establishes itself; the kernel should not rely on correct operation of the LSRs or externally-supplied meta-objects (the error system should be used to report bogus behavior of other parties). An LSR may also do internal consistency checking, relying on correct operation of the kernel (and of course any meta-objects the LSR itself supplies); `ilu_Check` may be used for such checks.

A.4.8 Debugging

`ilu_DebugPrintf` (see '`iluxport.h`') may be used to display messages to the user (if and when an LSR is in such dire straights that it needs to). Use this instead of printing directly to stderr to take advantage of the kernel's standard re-direction facility. See the "from debug.c" section of '`iluxport.h`'.

The kernel is loaded with conditional printf statements, for debugging purposes; an LSR may also include conditional printf statements using the same mechanism. See '`runtime/kernel/iludebug.h`', for the definition of the `DEBUG` macro and the particular bits available for control. The "from debug.c" section of '`iluxport.h`' also includes procedures that allow the control bits to be set; this should be exposed in your language mapping.

A.4.9 Fine Grain Time

The ILU kernel uses a representation of time that has subsecond resolution. The exact resolution is dependent on the OS, and can be read from a global variable. See the "Time" section of '`iluxport.h`'.

A.4.10 FD Budget

The ILU kernel will restrain itself to use at most a given number of File Descriptors (FDs). See the "FD & Connection Management" section of '`iluxport.h`'. Your mapping should expose this control to applications.

A.4.11 Supporting Multiple Languages in One Runtime

It's not your job to integrate language *L*'s runtime system with language *M*'s. But supposing someone has, your LSR for *L* should be able to co-exist with the LSR for *M*. The kernel is prepared to deal with multiple LSRs. Each LSR should call `ilu_RegisterLanguage`, to get a `ilu_LanguageIndex` for use in certain kernel calls. To support 'Collectible' objects, an LSR normally calls `ilu_SetGcClient`, providing an object that can be 'pinged' to determine whether or not a process is still alive. In the case of multiple LSRs within one process, only one LSR should do this. An LSR can determine if a GC client has already been set by calling `ilu_IsGcClientSet`.

A.4.12 Type Information

The stubs for an ISL interface are responsible for describing to the kernel each object type defined in that interface. The kernel will either make an internal copy of this information, or (if stubs for a different language have already described the type) check for equivalence of the given information with an already-existing internal copy. See the "Object Type Registry" section of '`iluxport.h`'.

Support for `ANY` introduces the need to describe all types defined in an interface. See ilutypes.h.

A.4.13 Simple Binding Service

This is a simple bootstrap for solving the name service problem. It should be used to get to a real name service. See the "Simple Binding" and "from sbilu.c" sections of 'iluxport.h'. Your mapping should expose this functionality.

A.4.14 Security Support

ILU provides hooks for the following three security services: authentication, message integrity, and message secrecy. The services are provided by a transport filter that's written against the GSS API (see the section on RPC and transport protocols in the refman). This filter is parameterized by certain identity and credential information. [More needs to be written about these parameters and their management --- Bill?] See the "Identities and Passports" section of 'iluxport.h'.

A.4.15 SBH schemes

An ILU SBH (string binding handle; see Section 1.3.1.6 [String Binding Handle], page 7) is organized like a WWW URL: a "scheme", followed by a colon and then some scheme-dependent stuff. The set of schemes understood by the kernel can be extended; see the "URL Syntax" section of 'iluxport.h'.

A.4.16 Pickle Support

A.4.16.1 The Kernel Interface to Pickle

The basic data structure the ILU kernel uses for pickles is `ilu_Pickle`, which is a struct containing some allocated storage. The basic calls for input, output, and sizing pickles are similar to those for other basic types:

```
extern void
  ilu_OutputPickle (ilu_Call, /* the call in progress */
    ilu_Pickle, /* pointer to the pickled value */
    ILU_ERRS((IoErrs)) *);

extern ilu_cardinal
  ilu_SizeOfPickle (ilu_Call, /* the call in progress */
    ilu_Pickle, /* pointer to the pickled value */
    ILU_ERRS((IoErrs)) *);

extern ilu_boolean /* the value read, PASS */
  ilu_InputPickle (ilu_Call, /* the call in progress */
   ilu_Pickle *, /* OUT:  the value input, PASS */
   ILU_ERRS((IoErrs)) *);
```

The ILU kernel exports a small set of functions to provide generic pickle support. They are intended to work with standard language-runtime functionality to provide language-specific pickle support. A pickle can be created via a function similar to this:

```
ilu_Pickle Pickle (ilu_string type_id, ilu_refany value)
{
  ilu_Call_s call;
  ilu_cardinal size;
  ilu_Pickle pickle = { 0 };
  ilu_Error err;

  ilu_StartPickle (&call, &err);
  if (ILU_ERRNOK(err)) return pickle;
  size = <size-fn> (&call, value, &err);
  if (ILU_ERRNOK(err)) return pickle;
  ilu_WritePickle (&call, size, type_id, &err);
  if (ILU_ERRNOK(err)) return pickle;
  <output-fn> (&call, value, &err);
  if (ILU_ERRNOK(err)) return pickle;
  ilu_EndPickle (&call, &pickle, &err);
  return pickle;
}
```

where <*size-fn*> is the normal function which calculates the marshalled size of a value, and <*output-fn*> is the normal function which causes the value of that type to be output. Of course, the example code must be re-written in the target language instead of in C.

Similarly, a language-specific value may be extracted from a pickle with the analogue of this code:

```
ilu_string Unpickle (ilu_Pickle pickle, ilu_refany *value)
{
  ilu_string type_id;
  ilu_Error err;
  <input-fn-type> <input-fn>;

  type_id = ilu_PickleType (pickle, &err);      /* read-only */
  if (ILU_ERRNOK(err)) return ILU_NIL;
  <input-fn> = <map-type-id-to-input-fn> (type_id);
  if (input_fn == NULLFN) { /* this type unknown in this LSR */
    ILU_CLER(*err);
    *value = ILU_NIL;
    return ILU_NIL;
  } else {
    ilu_Call_s call;
    ilu_Pickle p2;

    ilu_StartPickle (&call, &err);
    if (ILU_ERRNOK(err)) return ILU_NIL;
    if (!ilu_ReadPickle(&call, pickle, &err)) return ILU_NIL;
    *value = <input-fn> (&call, &err);
```

```
        if (ILU_ERRNOK(err)) return ILU_NIL;
        ilu_EndPickle(&call, &p2, &err);
        if (ILU_ERRNOK(*err)) {
          if (*value != ILU_NIL)
            <free-fn> (*value);
          return ILU_NIL;
        }
        ILU_CLER(*err);
        return type_id;
      }
    }
```

As before, the function <*input-fn*> is the normal language-specific input function for the type designated by `type_id`, and <*free-fn*> is the way of freeing an unused value. The value of `type_id` is read-only, and only guaranteed to exist for the lifetime of the pickle. The function <*map-type-id-to-input-fn*> must be provided by the language runtime; it will usually need some support from the language stubber.

A non-obvious side effect of calling `ilu_ReadPickle` is that control over the allocated data inside the *pickle* parameter is passed to the kernel, from the language runtime. That space may be reclaimed, as in the example above, by passing a non-NIL pickle reference to the final call to `ilu_EndPickle`; otherwise it will be freed in the call to `ilu_EndPickle`.

Finally, an `ilu_Pickle` value may be freed with a call to `ilu_FreePickle`.

See the file '*ILUSRC*`/runtime/kernel/ilutypes.h`' for more information on the kernel routines referred to here.

A.4.16.2 Pickles and the CORBA IIOP

Generally, pickles are sent over a wire protocol as a simple sequence of bytes, which consist of a prefix byte, a type ID, and the marshalled form of the value. However, the CORBA `IIOP` requires a full description of the type to be sent, rather than just the type ID. This makes using pickles with the `IIOP` much more expensive than with other protocols, but must still be supported for CORBA compliance.

To support this, the LSR must provide full type information to the kernel for each constructed type in the ISL interface. The LSR does this by calling type registration routines exported from the ILU kernel. Typically, these calls are made by stubber-generated interface initialization code. Some simple constructed types may be registered with a single call: `SEQUENCE`, `OPTIONAL`, `ARRAY`, `OBJECT` and alias types. Others (`RECORD`, `UNION`, `ENUMERATION`) require a sequence of calls. For example, a `RECORD` type is first registered by calling `ilu_RegisterRecordType`; the result of this call is then used in subsequent calls to `ilu_RegisterRecordField`, one call per field in the `record` type.

These calls should only be made if both VARIANT support and IIOP support are configured into the ILU system; the kernel will not export the required functions if one or the other is not available.

Appendix B The TIM Documentation Language

This document describes the TIM documentation language that the documentation for ILU is written in. It is not necessary to be familiar with TIM to use ILU; you will only need to know TIM if you wish to use it to write or modify documentation.

TIM is essentially a superset of the GNU texinfo language, version 2. It adds several features such as support for pictures and URLs, but its most important extension is to provide domain-specific markup commands to allow adding arbitrary meta-information to Texinfo documents. You should be familiar with the basic Texinfo system first. Documentation on Texinfo is supplied with the ILU distribution; you should be able to find it in the files '`ILUSRC/doc/texinfo2.ps`'.

B.1 Introduction

Both TIM and Texinfo input files contain text 'marked up' with document markup commands. These commands are similar to LaTeX commands, except that they start with an at-sign character rather than a backslash. They contain meta-information about the area of the text to which they apply. For example:

```
A kernel server @dfn{export}s its objects by making them available
to other modules.  It may do so via one or more @dfn{port}s, which are
abstractly a tuple of (@metavar{rpc protocol}, @metavar{transport type},
@metavar{transport address}).  For example, a typical port might
provide access to a kernel server's objects via @code{(@protocol{Sun RPC},
@transport{TCP/IP}, UNIX port 2076)}.  Another port on the
same kernel server might provide access to the objects via
@code{(@protocol{Xerox Courier}, @transport{XNS SPP}, XNS port 1394)}.
```

When formatted, this paragraph would look like

> A kernel server *export*s its objects by making them available to other modules. It may do so via one or more *port*s, which are abstractly a tuple of (*rpc protocol*, *transport type*, *transport address*). For example, a typical port might provide access to a kernel server's objects via `(Sun RPC, TCP/IP, UNIX port 2076)`. Another port on the same kernel server might provide access to the objects via `(Xerox Courier, XNS SPP, XNS port 1394)`.

There are two kinds of markup commands: without arguments or with arguments. The commands without arguments always span some portion of the document, so we call them *span commands*. They may be nested, but may not overlap. There are two forms of span commands, *style commands* and *format commands*. The style commands mark some section of the text, typically a short sequence of text, with a single attribute, which may be either a semantic tag like `important`, or a formatting style like `italic`. The format commands apply a similar tag to a block of the input; they begin with a single line containing `@`*attribute*, and end with a single line containing `@end` *attribute*. Style commands may be nested in a block command, but block commands should not be nested in style commands.

Markup commands with arguments always take a single line. The line begins with `@`*attribute*, followed by whitespace, followed by the arguments, separated by whitespace. If there is whitespace in an argument,

the argument is surrounded with braces, as in
`@deffun {struct foo} Bar ( arg )`

B.2 Extensions to GNU Texinfo

TIM removes the need to begin every file with `\input texinfo`, and to end every file with `@bye`. These lines are added automatically by TIM as needed. This allows a file to define both a stand-alone document, and to be included as a section in some larger document.

B.2.1 TIM Domain-Independent Format

TIM domain-independent format (*DIF*) is basically Texinfo with four new built-in commands. They are:

- `@url`, a style command, is used to mark World Wide Web URL forms that appear in the text.
- `@picture`, a command with arguments, is used to include an Encapsulated Postscript picture into the document. It takes two arguments, the name of the file, and a caption for the picture. The caption may be omitted.
- `@ttitalic`, a style command, is used to indicate that this span should be rendered in an italic typewriter font, if available.
- `@timmacro`, a command with arguments, allows the user to define domain-specific markup commands. The two arguments are the macro name, and the macro's replacement in vanilla DIF.

B.2.2 Defining Domain-Specific Markup Commands With `@timmacro`

[TBD]

B.3 TIM Tools

ILU provides a program called `tim` to turn TIM files into either PostScript, text, or GNU Info files. It is invoked as

`tim` *output-format* `[ -s` *flag* `] [ -m` *macros* `] [ -o` *output-file* `] [` *input-file*`.tim ]`

where *output-format* must be either **-p** for Postscript output, **-i** for GNU Info output, **-d** for TIM DIF output, **-t** for plain text output, and **-x** for vanilla GNU Texinfo output.

In addition, the switch **-v** can be specified to cause the `tim` script to output information about progress, the switch **-m** *macro-file-name* may be specified to have `tim` pre-load a file of `@timmacro` macros, the switch **-s** may be specified to set various TexInfo conditional flags, and the switch **-o** *output-file-name* may used to specify the output file. If no input file is specified, `tim` reads from the standard input. If no output file is specified, `tim` writes to the standard output.

`tim` is a script written in the Python script language, so you will need to have `Python` installed to use it. See the ILU installation instructions for a location from which `Python` can be FTP'ed. The script uses the GNU programs `texindex` and `makeinfo`, along with `TeX` and `dvips`, so it is necessary to have all four of those programs installed to use `tim`.

Another program called `timdif2html` can be used to turn TIM DIF files into World Wide Web HTML. See the end of the `timdif2html` script for instructions on how to use it. It in turn uses the script `eps2gif`, which requires having `ghostscript` built with a GIF driver. Both `timdif2html` and `eps2gif` are Perl scripts, so the Perl interpreter `perl` must be installed to use them.

B.4 Markup Commands used with ILU

The file '*ILUSRC*`/doc/ilu-macros.tim`' defines the following TIM markup commands that are used with the ILU documentation:

- `@var` is used to indicate a regular programming language variable. The term `@metavar` is used to mark meta-variables.

TIM also extends texinfo by adding the following markup:

- `@C` is used to mark artifacts of the C language, e.g., `@C{#define}`.
- `@C++` is used to mark artifacts of the C++ language, e.g., `@C++{#define}`.
- `@class` is used to mark names of object classes.
- `@command` is used to mark user input, such as a user-typed shell command, when it occurs in the normal flow of text. The term `@userinput` is used when the user input occurs within a `@transcript` section.
- `@codeexample` is used to mark code that is excerpted in the style of a texinfo `example`. The term `@codeexample` should appear on a line by itself, before the text of the code, and the terms `@end codeexample` should appear on a line by itself, at the end of the text of the code.
- `@cl` is used to mark artifacts of the Common Lisp language, e.g., `@cl{defmacro}`.
- `@constant` can be used to mark constant names and values that appear in the text.
- `@exception` is used to mark names of exceptions.
- `@fn` is used to mark function names that occur in the text.
- `@interface` is used to mark interface names.
- `@isl` is used to mark artifacts of the ILU ISL language, e.g. `@isl{SIBLING}`.
- `@java` is used to mark artifacts of the Java language, e.g. `@java{class Foo;}`.
- `@kwd` is used to mark keywords that occur in the text.
- `@language` is used to mark names of computer or human languages.
- `@m3` is used to mark artifacts of the Modula-3 language, e.g. `@m3{INTERFACE Foo;}`.
- `@macro` is used to mark names of macros that occur in the text.

- `@message` is used to mark in-line text that is a message a program may write to its output.
- `@metavar` is used to mark meta-variables.
- `@method` is used to mark method names.
- `@module` is used to denote module names for those languages which support them, such as Common Lisp package names, or Java package names.
- `@parm` is used to mark parameter names.
- `@picture` is used to include a file containing encapsulated Postscript of a diagram or picture. It should appear on a line, followed by the name of the file containing the picture, followed by a newline. We find the InterViews tool `idraw` works well in creating diagrams in the form of encapsulated Postscript.
- `@program` is used to mark program names that occur in the text.
- `@protocol` is used to mark names of ILU RPC protocols.
- `@symbol` is used to mark names of symbols in Makefiles or object files.
- `@system` is used to mark system names that occur in the text.
- `@switch` is used to mark command-line switches or options to programs.
- `@transcript` is used to mark an example that is a dialog between a user and a program. The term `@transcript` should appear on a line by itself, before the text of the dialog, and the terms `@end transcript` should appear on a line by itself, at the end of the dialog. The term `@userinput` may be used within a transcript.
- `@transport` is used to mark the names of ILU data transport systems.
- `@type` is used to mark the names of programming language types.
- `@url` is used to mark World Wide Web urls.
- `@userinput` is used to mark text typed by the user in a transcript section.

Appendix C The ILU Common Lisp Portable DEFSYSTEM Module

The ILU Common Lisp support uses files called 'sysdcl's to describe the generated lisp files for a particular interface. A *sysdcl* is similar to a UNIX 'Makefile', in that it describes the dependencies of the files of a module on each other. As part of ILU, we supply an implementation of a sysdcl interpreter, implemented in the `DEFSYSTEM` package (which is also nicknamed `PDEFSYS`). The notion is that to load a module, the user loads the sysdcl which describes it, then uses the `DEFSYSTEM` commands to compile and load the files of that module. The rest of this section describes this system in more detail. All symbols described here are in the `pdefsys` package unless otherwise specified.

pdefsys:set-system-source-file (*NAME* `string`) (*PATHNAME* `pathname`) Function

Informs the defsystem utility that the definition of the system name can be found by loading the file pathname.

pdefsys:load-system-def (*NAME* (`or symbol string`) &optional (*RELOAD* `boolean t`) => `boolean` Function

If there is a system named name and reload is false (the default), does nothing. Otherwise, loads the system defintion from a file. If `pdefsys:set-system-source-file` has been used to give an explicit source file for the system defintion, that file is used. Otherwise the file '*NAME*`-sysdcl.lisp`' is loaded from the directory specified in `pdefsys:*sysdcl-pathname-defaults*` if such a file exists. Returns false if the system was not loaded and is not already defined, true otherwise.

pdefsys:*sysdcl-pathname-defaults* Variable

Specifies the location for system declaration files. `*sysdcl-pathname-defaults*` is a list of pathnames; each location is searched for the declaration file. The default value is `(list #P"/import/commonlisp-library/sysdcl/")`.

pdefsys:defsystem (*NAME* `string`) (*SYSTEM-OPTIONS* `plist`) &rest (*MODULE-DESCRIPTIONS* `module-list`) Macro

The name of the system (which is interned in the current package), is used by defsystem to allow dependencies between multiple systems.

The *SYSTEM-OPTIONS* is a plist which may contain each of the following keywords:

- `:default-pathname ((or string pathname))`

  The default place in which to find files; this value defaults to the null string. This argument is evaluated (unlike most of the others).

- `:default-binary-pathname ((or string pathname))`

  The default location in which to place and look for binaries. This defaults to the value of the :default-pathname option. This argument is evaluated (unlike most of the others).

- `:default-package` (`(or symbol package)`)

  The default package to load/compile modules in; this value defaults to the current package.

- `:default-optimizations` (`list`)

  List of default compiler optimizations settings to use when compiling modules. If `nil`, optimization levels are not changed.

- `:needed-systems` (`list`)

  A list of subsystems; this value defaults to `nil`.

- `:load-before-compile` (`(or boolean list)`)

  A list of subsystems needed for compilation; this value defaults to `nil`. A value of `T` means all needed subsystems.

The module-descriptions is a list of modules which make up a system. A module is a list whose car is the module name and whose cdr is a list of keywords and values. The module keywords may contain each of the following:

- `:load-before-compile` (`list`)

  The load-before-compile keyword specifies a list of modules which will cause this module to be recompiled. If any of listed modules is newer then the current module; the current module will be recompiled. If the current module is recompiled the list of recompile dependencies will be loaded first.

  This is also a recursive recompilation. If foo dependends on bar and bar is out of date then bar will be recompiled before foo is recompiled.

  A value of `T` means all modules that occur earlier in the system definition. This value defaults to `nil`.

- `:load-after` (`list`)

  The load-after keyword specifies a list of modules which should be loaded before the current module is loaded. This option is useful only for modules during compilation since the load order will normally be satisfied during a load-system. A value of `T` means all modules that occur earlier in the system definition. This value defaults to `nil`.

- `:pathname` (`(or string pathname)`)

  The pathname keyword specifies a pathname to find the current module. Normally the pathname is the result of the concatenation of the default pathname for the system and the module name. This value defaults to `nil`. This argument is evaluated, unlike the other module options.

- `:binary-pathname` (`(or string pathname)`) Specifies the pathname for the binary of the current module. Defaults to the pathname with the same directory & name as the module source, with an appropriate file type.

- `:package` (`(or symbol package)`) The package keyword specifies a package in which to load/compile the current module. Normally the package is the default package for the system. This value defaults to `nil`.
- `:compile-satisfies-load` (`boolean`) The compile-satisfies-load keyword specifies that compiling the current module will satisfy a load (and hence the current module will not be loaded during a compile). This option is useful only for files containing macros. This value defaults to false.
- `:language` (`keyword`) The language the source is written in. See the variable `pdefsys:*language-descriptions*` for further info. The default is :LISP.
- `:optimizations` (`list`) List of compiler optimization settings to use when compiling the module. A useful value for lisp might be ((SPEED 3) (SAFETY 0)); for C ("-O"). If not present, the system's default-optimizations are used. If they too are absent, the current settings are used.
- `:libraries` (`list`) List of object libraries to load when the module is loaded. This is only useful for languages like C.
- `:features` (`list`) Run-time conditionialization, similar to #+. The module is used iff the features is "true" in the same way that #+ interpretes the features. Additionally, features may be `T` (the default) which is always true, or a list of features which is true iff at least one of the features matches.
- `:eval-after` (`form`) If present, a form that will be evaluated after the module is loaded. It should be noted that this is evaluated each time the module is loaded, whether or not the coresponding -file- is loaded.
- `:binary-only` (`boolean`) If true, declares that there is no source file associated with the module. No attempt will be name to compile it. Defaults to false.

pdefsys:*language-descriptions* Variable

An alist describing how files written in different languages are compiled and loaded. Each entry in the list is of the form (language-name source-file-type binary-file-type compile-fn load-fn). The language-name is the (keyword) name of the language. Source-file-type and binary-file-type are lists of strings; they are the file-types for source and binary files for the language. The compile-fn is symbol that will be called with three arguments to compile the source file; the pathname of the source file, the pathname of the binary output file, and a list of the optimizations declared for the module. Load-fn is a symbol that will be called with two required argument to load the binary file: the pathname of the binary, and a list of object library files to use.

The initial value of *language-descriptions* contains a description of `:lisp`, `:k&r-c` and `:ansi-c` languages. The description of `:lisp` uses the second argument to the compile-fn as a list of compiler optimization settings. The description of `:k&r-c` and `:ansi-c` uses the list as a set of additional arguments to pass to the C compiler.

pdefsys:undefsystem (*NAME* (`or symbol string`)) Macro

This macro removes the named system description from the list of all systems.

pdefsys:load-system (*NAME* (`or symbol string`)) &key (*RELOAD* `boolean nil`) (*RECURSE* `boolean nil`) (*TRACE* `boolean nil`) (*SOURCE-IF-NEWER* `boolean nil`) Function

This function loads the modules of the system with the specified name and is called recursively for all required systems. While the system is being loaded, the special variable `pdefsys:*current-system*` is bound to the name of the system.

The keyword args act as follows:

- *RELOAD*

  The reload keyword, if true, specifies that a full reload of all system modules and required systems, regardless of need. This value defaults to false.

- *RECURSE*

  If recurse is true, required systems are reloaded if the currently loaded version is not up-to-date or if the reload option is true. If recurse is false (the default), a required subsystem is not loaded if there is already a version loaded.

- *TRACE*

  If true, no module or subsystem is actually loaded. Instead a message is printed out informing you of what would have been loaded. The default value is false.

- *SOURCE-IF-NEWER*

  If true and a module's source is newer than its binary, or the binary does not exist, the source will be loaded. In all other cases, the binary will be loaded. The default value is false.

pdefsys:compile-system (*NAME* (`or string symbol`)) &key (*RECOMPILE* `boolean nil`) (*RELOAD* `boolean nil`) (*PROPAGATE* `boolean nil`) (*TRACE* `boolean nil`) (*INCLUDE-COMPONENTS* `boolean nil`) Function

This function compiles the modules of the system with the specified name and is called recursively for all required systems. While the system is being compiled, the special variable `pdefsys:*current-system*` is bound to the name of the system.

The keyword args act as follows:

- *RECOMPILE*

  The recompile keyword, if true, specifies that all modules should be recompiled, regardless of need. This value defaults to false.

- *INCLUDE-COMPONENTS*

  The include-components keyword, if true, specifies that compile-system should load all required systems. This value defaults to true.

- *RELOAD*

  The reload keyword, if true, specifies that a full reload of all system modules and required systems, regardless of need. This value defaults to false.

- *PROPAGATE*

  If true, the compile propagates to all subsystems (those required to load and to compile this system). The default is false.

- *TRACE*

  If true, no module of subsystem is actually compiled. Instead a message is printed out informing you of what would have been done. The default value is false.

pdefsys:show-system (*NAME* (`or string symbol`)) Function

This function outputs a formatted description of the system with the specified *NAME*.

C.1 Pathname Support

Some lisps don't yet support the structured directories specified in CLtL2 (p. 620). To support those lisps, `pdefsys` contains two functions which do support some of that functionality.

pdefsys:make-pathname &key *host device directory name type version defaults* Function

pdefsys:pathname-directory *pathname* Function

These functions shadow the functions in the `common-lisp` package, and support the subdirectory list syntax described as follows (From the X3J13 PATHNAME-SUBDIRECTORY-LIST proposal):

> It is impossible to write portable code that can produce a pathname in a subdirectory of a hierarchical file system. This defeats much of the purpose of the pathname abstraction.
>
> According to CLtL, only a string is a portable value for the directory component of a pathname. Thus in order to denote a subdirectory, the use of punctuation characters (such as dots, slashes, or backslashes) would be necessary. The very fact that such syntax varies from host to host means that although the representation might be "portable", the code using that representation is not portable.
>
> This problem is even worse for programs running on machines on a network that can retrieve files from multiple hosts, each using a different OS and thus different subdirectory punctuation.
>
> Proposal:
>
> Allow the value of a pathname's directory component to be a list. The car of the list is one of the symbols :ABSOLUTE or :RELATIVE. Each remaining element of the list is a string or a symbol (see below). Each string names a single level of directory structure. The strings should contain only the directory names themselves -- no punctuation characters.
>
> A list whose car is the symbol :ABSOLUTE represents a directory path starting from the root directory. The list (:ABSOLUTE) represents the root directory. The list (:ABSOLUTE "foo" "bar" "baz") represents the directory called "/foo/bar/baz" in Unix [except possibly for alphabetic case -- that is the subject of a separate issue].

A list whose car is the symbol :RELATIVE represents a directory path starting from a default directory. The list (:RELATIVE) has the same meaning as `nil` and hence is not used. The list (:RELATIVE "foo" "bar") represents the directory named "bar" in the directory named "foo" in the default directory.

Here's an sample sysdcl file that shows how the DEFSYSTEM functions and these pathname functions work together.

```
(in-package "DEFSYSTEM")

(defvar *my-system-default-directory*
  (make-pathname :directory
                 '(:absolute "import" "my-system" "release-1.0")))

(set-system-source-file :mysys-test
  (make-pathname :directory '(:relative "test")
                 :name "test-sysdcl"
                 :defaults *my-system-default-directory*))

(defsystem :my-system (:default-pathname *my-system-default-directory*
                       :default-package "USER"
                       :load-before-compile ()
                       :needed-systems ())
  ...)
```

Appendix D The ILU Common Lisp Lightweight Process System

D.1 Introduction

Although it is not required by the specification, most Common Lisp implementations include a facility for multiple, independent *threads* of control (often called *lightweight processes*) within a single Lisp environment. Unfortunately, this facility is not standardized across the various implementations. Although the capabilities provided are very similar across implementations, the details of lightweight processes and the interface to them differ significantly. This situation makes it difficult to write programs that use lightweight processes and yet are portable across Common Lisp implementations.

Common Lisp ILU does not requires lightweight processes in order to function, but they are useful. In particular, servers typically make heavy use of lightweight process facilities. The purpose of the ILU CL Process Interface is to provide a standardized, portable interface to lightweight processes for use within the ILU environment. This interface isolates ILU users from the differences in the various lightweight process implementations and allows them to write programs that are portable across all implementations to which the ILU CL Process Interface has been ported. At present, these implementations include Franz Allegro CL 4.1, and Lucid Common Lisp 3.0 (a.k.a., Sun Common Lisp).

This chapter explains how the ILU CL Process Interface works for ILU users. It begins with an overview that describes the ILU CL Process model, followed by a listing of some functional capabilities of this model. After brief discussions of the implementation architecture and general limitations of the ILU CL Process Interface, the chapter presents an example of how to use the interface to define a simple shared FIFO queue. Next, it lists all of the functions and macros necessary to use lightweight processes in the ILU environment. The chapter concludes with a brief list of references.

To use the information in this chapter, you should be familiar with Common Lisp and with the notion of processes and threads in an operating system. Familiarity with the UNIX process model would also be helpful. (See the References section for recommendations on further reading.)

D.2 Overview Of The ILU CL Process Model

The ILU CL Process Interface features an interface to lightweight processes similar to that on the Symbolics Lisp machine. In particular, within a single Lisp environment (which on stock hardware runs as a single heavyweight UNIX process) there are multiple threads of control that can be scheduled independently. These threads are called *lightweight processes* (or sometimes just *processes*). Each lightweight process contains its own run-time control and binding stack, but it shares the global data and program address space with all other processes in the Lisp environment. Note that this arrangement differs from that of the UNIX heavyweight process facility, where each process has its own address space as well as its own run-time stack.

D.2.1 The Scheduler Process

Each lightweight process represents an independent thread of control. The multiple threads within the Lisp environment are managed by a special *scheduler* process. The ILU CL Process Interface makes no assumptions about the nature of this scheduler process. However, most implementations use a time-slice, priority-based scheduler. In such a scheduler, an interrupt occurs once every so often (called the scheduler's *quantum*). When the interrupt occurs, the process that is currently running is stopped and its state is saved. The scheduler then examines all processes that are runnable (that is, waiting to run) and restarts the process that has the highest priority. This process runs until the next interrupt or until it gives up control to the scheduler, whichever comes first. At any given time, the one process that is ''currently'' running is known as the *current process*.

D.2.2 States Of Processes

In the ILU CL Process model, each lightweight process is represented by a single Lisp object that maintains the information about that process. Also, each process is always in one of three states: *active*, *inactive*, or *killed*. A process maintains two lists of objects called, respectively, the *run reasons* and the *arrest reasons* for the process. For a process to be active, it must have at least one run reason and no arrest reasons. A process with no run reasons or at least one arrest reason is considered inactive. The ILU CL Process Interface provides functions for adding and removing run and arrest reasons for a process. Thus, the user (or a program) can move a process between the active and inactive states.

The scheduler runs only active processes. Until an inactive process is reactivated, it cannot run. A killed process is one that has been explicitly killed (using the `ilu-process:process-kill` function). A killed process can never be run again (that is, it can never be made active).

An active process can in turn be in one of two substates: *runnable* and *waiting*. A runnable process is ready to be restarted by the scheduler, which determines whether and when a process will actually be restarted based on its status (that is, priority) and the status of the other runnable processes. A waiting process is a process that has a *wait function* and a list of *wait arguments*. These two items are supplied to the process using the `ilu-process:process-wait` function. Periodically, the scheduler will `apply` the process's wait function to its wait arguments (in the context of the scheduler). If the result is a non-`nil` value, the wait function and wait arguments are removed from the process, and the process thereby becomes runnable. Usually, the scheduler evaluates the wait functions for all waiting processes every time around the scheduler loop. Therefore, it is important that wait functions be fast and very efficient.

D.2.3 Removing Or Killing Processes

You can reversably remove a process from a runnable state either by entering a wait or by making it inactive. In general, it is more efficient to make a process inactive because this removes it from the scheduler's active process list. Thus, the scheduler does not incur the cost of periodically evaluating its wait function. However, an inactive process cannot make itself active. It must depend on some other process to recognize when it is ready to run again and to reactivate it at that time. Although a waiting process is initially more costly than an inactive one, it is automatically returned to a runnable state by the scheduler whenever its wait function returns non-`nil`. Hence, no second process is needed to *restart* a waiting function. Thus, the choice between waiting a process and rendering it inactive depends on the architecture of the application being written.

When a process is first started, it is given a Lisp function and a set of arguments to this function. These are known as the process's *initial-function* and *inital-arguments*, respectively. A newly created process, `applys` its initial-function to its inital-arguments. When the initial-function returns, the process is automatically killed. Once killed it can never be restarted. You can also kill the process before the inital-function returns using the `ilu-process:process-kill` function, which causes the process to execute a `throw` in its current context. This `throw` causes the stack to unwind (executing unwind-protect forms along the way) and the initial-function to return, thereby killing the process.

D.2.4 Properties Of Processes

Every process has a number of properties. Specifically, a process has an arbitrary *process name* that identifies it in displays and in certain operations. Process names need not be unique. A process also has a *priority* that the scheduler uses optionally to determine when to schedule the process. Priorities are small integers and default to zero (0). In most implementations, processes with higher priorities are given scheduling preference. Negatives are used to indicate that a process should run as a background task when nothing else is running. Finally, a process has a *quantum*, which is the amount of time (measured in seconds) that the process wishes to run each time before it is interrupted. In some implementations, the scheduler uses a process's quantum to help determine the actual length of the time-slice given to the process. Many implementations ignore the quantum altogether.

D.2.5 Process Locks

The ILU CL Process Interface also includes a facility called *process locks* that supports exclusion-based sharing of a common resource (that is, a common object or data structure) or a critical region of code by two or more concurrent processes. A process lock is an object that a process can *lock* in order to claim exclusive access to the shared resource corresponding to the lock. Process locks are essentially a semaphore mechanism specialized for use with the ILU CL Process interface.

Each process lock has a name and a locker. A lock's name is for display purposes only. Processes can ask to gain or relinquish exclusive rights to the lock (called *locking* and *unlocking* the lock, respectively). While a process has rights to the lock, the lock's locker is (generally) the process object for that process. When a process asks to lock a lock that is already locked, the asking process blocks and ‘‘waits’’ until the lock is free. Waiting does not necessarily use the standard wait mechanism. Some implementations use process deactivation to implement the ‘‘wait’’ in this case. Some implementations may also maintain a queue of processes waiting for a lock to be freed, thereby ensuring fair access to the lock. Other implementations may not maintain such a queue, and therefore fair access to the lock is not guaranteed.

Process locks are contractual in nature. The various processes sharing a resource (or critical section of code) must all agree not to access the common resource while the process lock corresponding to that resource is held by another process. Furthermore, they must agree to lock the process lock whenever they need exclusive access to the resource, thereby notifying the other processes of their intent. Moreover, the correspondence between a process lock and the shared resource is a matter of agreement between the cooperating processes. The system does not provide any direct support for this correspondence (although it may be added on at a higher level built on top of the basic process lock mechanism).

Process locks provide a code-centered ‘‘sharing’’ mechanism where the access control is built into the programs that access the shared resource. Process locks are suited for closed, or non-extensible, applications where the shared resource is a standard Lisp data structure (that is, not a CLOS object) and where efficiency is a major concern. For applications not meeting these criteria, a mechanism in which a CLOS object itself controls simultaneous access to its internal data structures may be more appropriate.

D.3 Functional Overview

The ILU CL Process Interface provides all of the functions and macros necessary to use lightweight processes in the ILU environment. The functionality provided by these functions and macros includes:

- Starting new processes and killing processes
- Displaying status information, such as the current process, all active processes, or all known processes
- Accessing and modifying the properties of a process (for example, its name or priority)
- Adding/Removing arrest and run reasons for a process
- Allowing a process to give up control to the scheduler or enter into a wait state
- Temporarily turning off the scheduler so that the current process cannot be interrupted
- Creating, locking, unlocking, and modifying process locks

D.4 Implementation Architecture

The ILU CL Process Interface is implemented as a veneer over the existing process interfaces for a number of Common Lisp implementations (currently Franz Allegro CL and Lucid Common Lisp). In many cases, the implementation’s functions are simply imported and then exported from the `ilu-process` package. In other cases, a new function is wrapped around the implementation’s native function to change the name, arguments, or semantics of the function so that they match those required by the ILU CL Process Interface specification. In a few cases, whole new functions have been written to achieve functionality not provided by the original implementation.

The nature of the process object in the ILU CL Process Interface is not specified. The process object is inherited from the underlying implementation and may therefore be a list, a structure, a flavor object, or even a CLOS object. Because of this lack of specification, process objects cannot be specialized. Moreover, they cannot be accessed or modified in any way other than through the functional interface described in this chapter.

D.5 General Limitations

The ILU CL Process Interface assumes that the scheduler is loaded and running in the ILU environment. Procedures for starting the scheduler are not included in the ILU CL Process Interface. Some implementations, however, may require you to actually load and start up the scheduler. For example, in Franz Allegro CL, you need to evaluate `(mp:start-scheduler)` either at the top-level or in your '`.clinit.cl`' file in order to load and start up the scheduler.

The ILU CL Process Interface is subject to all of the limitations of its underlying implementations. In particular, one problem with most Common Lisp implementations on stock hardware is that the smallest scheduler quantum possible is one second. This means that each process gets to run for one second uninterrupted. For applications that involve real-time response, waiting for one second before an event can be handled is problematic. In practice, this problem can be lessened if all processes release control to the scheduler at regular, short intervals (that is, each few times around a tight inner loop), thereby making the effective quantum significantly less than one second. Note that this practice effectively reduces the scheduler to a prioritized, cooperative scheduler rather than the preemptive scheduler intended.

Most Common Lisp implementations build their process mechanism on top of a very powerful mechanism called *stack groups*. Stack groups provide for alternative run-time stacks in the Lisp environment that can be used for various purposes beyond implementing processes. For example, stack groups are an ideal substrate for implementing co-routines. Unfortunately, not all implementations provide an interface to stack groups (if indeed they have stack groups). Hence, an interface to stack groups is not a part of the ILU CL Process Interface.

D.6 How To Use The ILU CL Process Interface

The ILU CL Process Interface is intended as a programmer's interface; the functions and macros provided should be used to implement programs that run in the ILU environment. Although you can use any of the functions and macros directly from a Lisp listener, the interface is not designed particularly well for interactive use. The two exceptions to this rule are the functions `ilu-process:show-process` and `ilu-process:show-all-processes`, both of which are designed to print out status information in the Lisp listener window. Because it is a user-oriented function, `ilu-process:show-process` accepts either the process name or a process object to identify the process whose status is to be displayed.

Most implementations include an interactive interface to multiple processes and the scheduler. For example, Franz Allegro CL has a special top-level command language that is operative in every Lisp listener. This command language includes the following commands that deal specifically with lightweight processes (see Chapter 4 of *[Franz-92]* for more information):

- `:processes`

  Lists all processes (see `ilu-process:all-processes`)
- `:kill`

  Kills a process (see `ilu-process:process-kill`)

- `:arrest`

  Adds an arrest reason to a process (see `ilu-process:add-arrest-reason`)
- `:unarrest`

  Removes any arrest reason that was added to a process by `:arrest` (see `ilu-process:process-revoke-arrest-reason`)
- `:focus`

  Performs an `:arrest` on a process and arranges for all user keyboard input to be sent to the arrested process (usually to the debugger).

D.7 How To Program The ILU CL Process Interface

The following example illustrates how to use the ILU CL Process Interface to define a shared FIFO queue. Two processes will utilize this queue. A producer process will read input items from the user and place them on the shared queue. A consumer process will wake up every five seconds and read items from the shared queue, printing them on the standard output stream as they are taken off the queue. Access to the shared queue will be controlled using a process lock associated with the queue.

```
;;;______________________________________________________
;;; the shared queue, its process-lock, and its accessors/mutators
;;;

(defvar queue (list t) "The shared queue")

(defvar queue-lock (ilu-process:make-process-lock :name "queue lock")
  "process lock for queue")

(defun queue-pop (queue)
  "Pop an item off of the shared FIFO queue.
   Use ilu-process:with-process-lock to prevent collisions between processes.
  "
  (ilu-process:with-process-lock (queue-lock)
    (prog1
(cadr queue)
      (rplacd queue (cddr queue)))
    ))

(defun queue-push (queue item)
  "Push an item onto the shared FIFO queue.
   Use ilu-process:with-process-lock to prevent collisions between processes.
  "
  (ilu-process:with-process-lock (queue-lock)
    (nconc queue (list item))
    ))

(defun queue-empty-p (queue)
  "Is queue empty?
```

```
   Use ilu-process:with-process-lock to prevent collisions between processes.
  "
  (ilu-process:with-process-lock (queue-lock) (null (cdr queue)) ))

;;;_________________________________________________
;;;  The producer function
;;;

(defun produce ()
  "Loop reading an item from the user and pushing it onto the shared queue."
  (let (Item)
    (loop
      ;; Wait until there is something on the input stream.
      (ilu-process:process-wait "Waiting for input" #'listen *standard-input*)

      ;; Read the input.
      (setq Item (read *standard-input*))

      ;; Check to see if it is the EOF marker and exit if so.
      (when (eq Item :EOF) (return nil))

      ;; Push the item onto the queue.
      (queue-push queue Item)
      )))

;;;_________________________________________________
;;; The consumer function
;;;

(defun consume ()
  "Wake up every five seconds and see if there is something on the shared
   queue.  If there is, pop it off and print it on standard output.
   If the queue is empty and the producer process is not alive, terminate.
  "
  (loop
    ;; Check to see if there is anything on the queue.
    (if (not (queue-empty-p queue))
;; There is an item on the queue; pop and print all items.
(do ()((queue-empty-p queue))
  (fresh-line t)
  (princ "Output: ")
  (prin1 (queue-pop queue))
  (fresh-line t)
  (finish-output t))

      ;; Queue is empty; check to see if the producer is still alive.
      (if (null (ilu-process:find-process "Producer Process"))

  ;; Producer not alive; terminate.
  (return nil)))
```

```
    ;; Sleep for five seconds; this gives up control immediately
    ;; so some other process can run.
    (sleep 5)
    ))

;;;________________________________________________
;;;   Main function; starts consumer and producer processes
;;;

(defun test-queue ()
  "Start consumer and producer processes. Wait in an idle loop until
   both the producer and the consumer processes die.  This function is
   meant to be evaluated in the Lisp listener.  Waiting until both
   processes die ensures that the Lisp listener does not interfere
   with user input to the producer.
  "
  (let (Producer Consumer)
    ;; Start the producer first; the consumer needs the producer to run.
    (setq Producer (ilu-process:fork-process "Producer Process" #'produce))
    ;; Start the consumer.
    (setq Consumer (ilu-process:fork-process "Consumer Process" #'consume))
    ;; Show processes on the standard ouput.
    (ilu-process:show-all-processes)
    ;; Wait until both consumer and producer are dead.
    (ilu-process:process-wait "Waiting for godot"
#'(lambda (P1 P2)
    (not
     (or (ilu-process:process-alive-p P1)
 (ilu-process:process-alive-p P2))))
Consumer Producer)
    ))
```

The following is a transcript of this test program in operation:

```
;;;________________________________________________
#73: (test-queue)
-------------Data on all processes follows---------
Process: "Consumer Process"
  Process-alive-p: T
  Process-active-p: T
  Process-quantum: 2
  Process-priority: 0
  Process-run-reasons: (:START :START)
Process: "Producer Process"
  Process-alive-p: T
  Process-active-p: T
  Process-quantum: 2
  Process-priority: 0
  Process-run-reasons: (:START :START)
```

```
Process: "Initial Lisp Listener"
  Process-alive-p: T
  Process-active-p: T
  Process-quantum: 2
  Process-priority: 0
  Process-run-reasons: (:ENABLE)
123
Output: 123
456
789
Output: 456
Output: 789
444
555
666
Output: 444
Output: 555
Output: 666
:eof
NIL
#74:
```

D.8 The ILU CL Process Interface

The following sections detail the functions and macros that make up the ILU CL Process Interface. All are assumed to be in the `ilu-process` package unless otherwise specified. Arguments are shown with their type, if they have any restrictions on their type. Return types are shown if the function returns a value. Optional arguments are shown with their type and their default value.

D.8.1 The Process Object

The following listings describe the object that is used to represent each lightweight process.

i lu-process:process Type

A Lisp object representing a single process. This object is to be used only as a handle for the process. To alter the state or characteristics of a process, use the external function interface defined below. The exact nature of the process object differs between implementations. In particular, it may or may not be a flavor or a CLOS object. Hence, it is not safe to specialize processes.

find-process (*NAME* `string`) => `process` Function

Returns the process object whose name is *NAME*. Only `ilu-process:process-alive-p` processes (that is, processes on the list returned from `ilu-process:all-processes`) are searched. This function returns `nil` if there is no matching process.

processp *OBJECT* => `boolean` Function

Returns non-`nil` if *OBJECT* is an object of type `process` for this implementation. This function returns `nil` otherwise.

D.8.2 Querying The Status Of The Scheduler And All Processes

The following functions and macros provide status information about the general state of processes and the scheduler in the Lisp environment.

active-processes => `list` Macro

Returns a list of all active processes; that is, processes that have at least one run reason and no arrest reasons. Note, however, that these processes are not necessarily runnable because they may be in a process-wait.

all-processes => `list` Macro

Returns a list of all processes currently known by the scheduler, including active and inactive processes but not processes that have been killed.

current-process => `process` Macro

Returns the process object for the current thread of control.

show-all-processes &optional (*STREAM* `streamp cl:*standard-output*`) (*VERBOSE* `boolean nil`) Function

Displays information about all processes known by the scheduler (that is, the processes returned by `ilu-process:all-processes`). Output is to *STREAM*, which defaults to the value of `cl:*standard-output*`. This function shows only non-`nil` fields unless *VERBOSE* is non-`nil`; the default is `nil`.

D.8.3 Starting And Killing Processes

fork-process (*NAME-OR-KEY-LIST* (`or string proplist`) (*FUNCTION* `function`) &rest *ARGS* => `process` Function

Creates a new process and returns the `process` object for this process. In this process, *FUNCTION* is `apply`ed to *ARGS*. If *FUNCTION* ever returns, the process is automatically killed. The *FUNCTION* is known as the initial-function of the process (see `ilu-process:process-initial-form`).

The new process is activated by default, although you can create it in a deactivated state by giving it a run reasons list with a value of `nil` or by giving it one or more arrest reasons as detailed below.

NAME-OR-KEY-LIST is either a string, in which case it serves as the name of the process, or it is a property list with one or more of the following property-value pairs:

- `:name` (`string`)

  A string to be used as the name of the process.

- `:priority` (`integer`)

  Sets the priority of the process to the given value (see `ilu-process:process-priority`).

- `:quantum` (`(or numberp nil)`)

  Sets the quantum of the process to the given value (see `ilu-process:process-quantum`). Defaults to `1`.

- `:stack-size` (`fixnum`)

  Sets the stack-size of the process (if possible in this implementation).

- `:run-reasons` (`list`)

  Sets the run reasons of this process to the given list. Unless run-reasons is non-`nil`, the forked process does not run until a `ilu-process:process-add-run-reason` is done. This property defaults to `(quote (:start))`.

- `:arrest-reasons` (`list`)

  Sets the arrest reasons of this process to the given list. If arrest-reasons is non-`nil`, the forked process does not run until a `ilu-process:process-revoke-arrest-reason` is done. This property defaults to `nil`.

- `:bindings` (`list`)

  A list of bindings (as in let) that are done in the context of the forked process before the function is run. This property defaults to `ilu-process:*default-process-bindings*`.

process-kill (*PROCESS* `process`) Function

Terminates *PROCESS* and removes it from the scheduler's consideration altogether. It is an error if *PROCESS* is not `ilu-process:processp` and `ilu-process:process-alive-p`.

A process may not terminate immediately. In particular, the process is first activated and scheduled. It is then forced to `throw` out of its initial-function, thereby properly unwinding and executing any unwind forms.

A killed process cannot be reactivated.

D.8.4 Waiting A Process

process-wait (*WHOSTATE* `string`) (*FUNCTION* `function`) &rest *ARGS* Function

The current process is suspended until *FUNCTION* applied to *ARGS* returns non-`nil`. During this time, the process's whostate (see `ilu-process:process-whostate`) is set to *WHOSTATE*.

Note that the current process is not deactivated. It is simply not scheduled to run until its wait-function returns non-`nil`. The scheduler re-evaluates the wait-function periodically. In general, the re-evaluation occurs whenever the waited process would be scheduled to run if it were not suspended. However, in some implementations it is run during every scheduler break.

D.8.5 Activating And Deactivating Processes

process-add-arrest-reason (*PROCESS* `process`) *OBJECT* — Function

Adds *OBJECT* to the list of arrest reasons for *PROCESS*. The *OBJECT* argument can be any Lisp object. It is an error if *PROCESS* is not `ilu-process:processp`.

Adding an arrest reason may cause a process to become deactivated. In particular, if this is the first arrest reason, then the process becomes deactivated (if it was previously activated).

process-add-run-reason (*PROCESS* `process`) *OBJECT* — Function

Adds *OBJECT* to the list of run reasons for *PROCESS*. The *OBJECT* argument can be any Lisp object. It is an error if *PROCESS* is not `ilu-process:processp`.

Adding a run reason may cause a process to become activated. In particular, if there are no arrest reasons and the added run reason is first, the process goes from a deactivated state to an activated state.

process-arrest-reasons (*PROCESS* `process`) `=> list` — Function

Returns the list of arrest reasons for *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`.

process-disable (*PROCESS* `process`) — Function

Causes *PROCESS* to become inactive by removing all of its arrest reasons and all of its run reasons. It is an error if *PROCESS* is not `ilu-process:processp`.

process-enable (*PROCESS* `process`) — Function

Causes *PROCESS* to become active by removing all of its arrest reasons and all of its run reasons and then giving it a single run reason (usually `:enable`). It is an error if *PROCESS* is not `ilu-process:processp`.

process-revoke-arrest-reason (*PROCESS* `process`) *OBJECT* — Function

Removes *OBJECT* from the list of arrest reasons for *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. *OBJECT* is compared to the existing arrest reasons using an `eq` test.

Revoking an arrest reason may cause a process to become activated. In particular, when the last arrest reason for a process is removed, the process is (re-)activated if it has at least one run reason.

process-revoke-run-reason (*PROCESS* `process`) *OBJECT* Function

Removes *OBJECT* from the list of run reasons for *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. The *OBJECT* argument is compared to the existing run reasons using an `eq` test.

Revoking a run reason may cause a process to become inactive. In particular, when the last run reason for a process is removed, the process is made inactive (if it was previously activate).

process-run-reasons (*PROCESS* `process`) Function

Returns the list of run reasons for *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`.

D.8.6 Accessing And Modifying The Properties Of A Process

process-active-p (*PROCESS* `process`) `=> boolean` Function

Returns non-`nil` if *PROCESS* is an active process object; that is, a process with no arrest reasons and at least one run reason. Otherwise, this function returns `nil`. It is an error if *PROCESS* is not `ilu-process:processp`.

process-alive-p (*PROCESS* `process`) `=> boolean` Function

Returns non-`nil` if *PROCESS* is alive (that is, has been created but has not been killed). Essentially, a process is alive if it is on the list returned by `ilu-process:all-processes`. It is an error if *PROCESS* is not `ilu-process:processp`.

process-initial-form (*PROCESS* `process`) `=> consp` Function

Returns the initial-form of the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. Note that the returned value is not an `eval`able form. It is merely the `cons` of the process's intial function onto a list of the initial arguments passed to the function. (See `ilu-process:fork-process`.)

process-name (*PROCESS* `process`) `=> string` Function

Returns the name of the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. The `ilu-process:process-active-p` function can be used with `setf` to change the name of a process.

process-priority (*PROCESS* `process`) `=> integer` Function

Returns the scheduling priority for the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. The `ilu-process:process-priority` function can be used with `setf` to change the priority of a process.

When the priorities are set, a small integer is generally used. Process priorities default to zero (0). Processes with higher priorities are given scheduling preference. Priorities can be negative if a process should run as a background task when nothing else is running.

Note that an implementation is free to ignore process priorities. Setting a process's priority is merely advisory. For this reason, the value returned by `ilu-process:process-priority` may not match the most recent `setf` on `ilu-process:process-priority`.

process-quantum (*PROCESS* `process`) => (`or numberp nil`) Function

Returns the quantum, which is the amount of time the scheduler allows a process to run each time its is rescheduled, for the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. The `ilu-process:process-quantum` function can be used with `setf` to change the quantum of a process.

The quantum is measured in seconds (not necessarily integral).

Note that an implementation is free to ignore process quantums. Setting a quantum is merely advisory. For this reason, the value returned by `ilu-process:process-quantum` may not match the most recent `setf` on `ilu-process:process-quantum`.

The default process quantum is 1.

process-wait-args (*PROCESS* `process`) => `list` Function

Returns a list of the arguments being passed to the wait-function of the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. (See `ilu-process:process-wait`.)

process-wait-function (*PROCESS* `process`) => (`or functionp nil`) Function

Returns the wait-function of the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. (See `ilu-process:process-wait`.)

process-whostate (*PROCESS* `process`) => `string` Function

Returns the whostate of the `process` object *PROCESS*. It is an error if *PROCESS* is not `ilu-process:processp`. The `ilu-process:process-whostate` function can be used with `setf` to change the whostate of a process. (See also `ilu-process:fork-process` and `ilu-process:process-wait`.)

show-process &optional (*PROCESS* `process`) (*STREAM* `streamp cl:*standard-output*`) (*VERBOSE* `boolean nil`) Function

Displays information about process *PROCESS*, which may be a `process` object or the name of a process known to the scheduler. If *PROCESS* is a symbol, it is downcased and converted to a string. *PROCESS* defaults to the current process. Output is is to *STREAM*, which defaults to the value of `cl:*standard-output*`. If *VERBOSE* is `nil` (defaults to non-`nil`), then only non-`nil` fields are displayed and the process's initial-form is not shown.

D.8.7 Miscellaneous Process/Scheduler Functions And Macros

process-allow-schedule Function

Suspends the current process and returns to the scheduler. All other processes of equal or higher priority have a chance to run before control returns to the current process.

process-interrupt (*PROCESS* `process`) (*FUNCTION* `function`) &rest *ARGS* Function

Forces *PROCESS* to apply *FUNCTION* to *ARGS* when it is next scheduled. When *FUNCTION* returns, *PROCESS* resumes execution where it was interrupted.

In general, `ilu-process:process-interrupt` is run immediately (that is, when *PROCESS* is next scheduled) if *PROCESS* is active, even if *PROCESS* is a process-wait. If *PROCESS* is not active, `ilu-process:process-interrupt` may wait until *PROCESS* is reactivated before *FUNCTION* is executed.

without-scheduling &body *BODY* Macro

Evaluates the forms in *BODY* with scheduling turned off. While the current-process is within the scope of `ilu-process:without-scheduling`, no other process will run. However, scheduling may be resumed if a `ilu-process:process-wait` or `ilu-process:process-allow-schedule` is executed within the scope of `ilu-process:without-scheduling`. Most Common Lisp I/O functions as well as the function `sleep` usually call some form of `ilu-process:process-allow-scheduling` and hence will resume scheduling if called within the scope of a `ilu-process:without-scheduling`.

D.8.8 Process Locks Interface

i lu-process:process-lock Type

The process lock object. You should access fields of this lock using only the functional interface listed in this section.

make-process-lock &key (*NAME* `(or nil string) nil`) => `process-lock` Function

Creates and returns a `process-lock` object with *NAME* as the name of the lock.

process-lock (*LOCK* `ilu-process:process-lock`) &optional (*LOCK-VALUE* `process (ilu-process:current-process)`) (*WHOSTATE* `(or nil string) <undefined>`) Function

Grabs *LOCK*, entering *LOCK-VALUE* as the lock's locker. *LOCK-VALUE* defaults to the current process. It is an error if *LOCK* is not `ilu-process:process-lock-p`.

If *LOCK* is already locked, then the current process waits until it is unlocked. The waiting is done using `ilu-process:process-wait`. The *WHOSTATE* argument is a string that is used as the whostate of the process if the process is forced to wait; defaults to an implementation-dependent string.

process-lock-locker (*LOCK* `ilu-process:process-lock`) => `t` Function

Returns the current locker of *LOCK*. It is an error if *LOCK* is not `ilu-process:process-lock-p`. This function returns `nil` if *LOCK* is currently unlocked, that is, has no locker. This value is *not* setfable. You should use `ilu-process:process-lock` to set the locker.

process-lock-name (*LOCK* `ilu-process:process-lock`) => (`or nil string`) Setf-able Function

Returns the name associated with *LOCK*. It is an error if *LOCK* is not `ilu-process:process-lock-p`. The `ilu-process:process-lock-locker` function can be used with `setf` to change the name of a process lock.

process-lock-p *OBJECT* => `boolean` Function

Returns non-`nil` if *OBJECT* is a `ilu-process:process-lock`. Otherwise, this function returns `nil`.

process-unlock (*LOCK* `ilu-process:process-lock`) &optional (*LOCK-VALUE* `t (ilu-process:current-process)`) (*ERROR-P* `boolean nil`) Function

Releases *LOCK*. It is an error if *LOCK* is not `ilu-process:process-lock-p`.

If *LOCK*'s locker is not `eq` to *LOCK-VALUE*, which defaults to the current process, then an error is signalled unless *ERROR-P* is `nil` (it defaults to `t`).

with-process-lock (*LOCK* `ilu-process:process-lock`) &key (*NORECURSIVE* `boolean nil`) &body *BODY* Macro

Locks *LOCK* for the current process and evaluates the forms in *BODY*. It is an error if *LOCK* is not `ilu-process:process-lock-p`.

If *NORECURSIVE* is `t` (the default), and if the current process already owns the lock (determined dynamically), then no action is taken. If *NORECURSIVE* is non-`nil`, then an error is signalled if the current process already owns *LOCK*.

If *LOCK* is held by another process, then the current process waits as in `ilu-process:process-lock`, which is described earlier in this section.

D.9 Handling Errors

Errors in most of the process functions will cause a break. There are no special tricks to handling these errors.

The `:focus` command is an important tool for using the Allegro CL debugger in a multiple-process environment. In particular, in Allegro CL a new process by default shares its standard input/output (I/O) stream with the Lisp listener. Generally, this is not a problem because the process runs in the background and does no I/O. However, if the process enters a break, the debugger needs to use the process's standard I/O stream to interact with the user. This could lead to a problem because the debugger I/O from the broken process will be interleaved with the Lisp listener's normal I/O. Specifically, the system will not be able to determine to which process user input is directed.

To avoid this situation, Allegro CL has the notion of a *focus process*. Input coming from the shared Lisp listener I/O stream is always sent to the focused process. Usually this is the Lisp listener process. However, if a background process breaks, you can use the `:focus` command to focus on the broken process and allow you to send input to the debugger running in that process. When the debugging is complete, `:focus` is automatically returned to the Lisp listener process.

The following transcript illustrates the use of the `:focus` command in Allegro CL:

```
;;;___________________________________________________________________________
;;; Start out focused on the Lisp listener process. List all processes.
<cl 71> :processes
"Initial Lisp Listener" is active.
;;;___________________________________________________________________________
;;; Second, start a test process that will enter a break immediately.
<cl 72> (ilu-process:fork-process "test" #'error "test break")
#<process test  #x13e92b1>
<cl 73>
;;;______________________________________________
;;; Process test enters a break.

Error: test break

;;; Still speaking to the Lisp listener process, list the processes.
[1] <cl 1> :processes
"test" is waiting for terminal input.
"Initial Lisp Listener" is active.
<cl 74>
;;;______________________________________________
;;; Now refocus on the test process.
<cl 74> :focus "test"
Focus is now on the "test" process.
;;;  Look at stack of test process.
<cl 75> :zoom
Evaluation stack of process "test":
 ->(EXCL::STM-SY-READ-CHAR #<synonym stream for *TERMINAL-IO*  #x13e9619>)
   (PEEK-CHAR NIL #<synonym stream for *TERMINAL-IO*  #x13e9619> NIL :EOF
NIL)
   (ERROR "test break")
   (ILU-PROCESS::PROCESS-INITIALIZATION-FUNCTION NIL
      #<Function ERROR  #x219ab9> ("test break"))
;;;______________________________________________
;;; Kill the test process (which is the current process).
<cl 76> :kill
Do you really want to kill process "test" [n]? y

;;; Automatic refocus to Lisp listener. Ask listener to list all processes.
Focus is now on the "listener" process.
```

```
<cl 77> :processes
"Initial Lisp Listener" is active.
```

For more information on the Lisp listener interface and the Lisp debugger, see the manual for the implementation of Common Lisp that you are using. For Allegro CL, refer to chapters 4 and 5 of *[Franz-92].*

D.10 Notes

It is possible for a process to do a non-blocking attempt to lock a process lock using the following idiom:

```
   (ilu-process:without-scheduling    ; Make sure this is not interrupted.
    (if (ilu-process:process-lock-locker LOCK) ; Is lock free?
        (ilu-process:process-lock LOCK)))      ; Lock is free, grab it.
   (if (eq                            ; Did we get the lock for this process?
        (ilu-process:process-lock-locker LOCK)
        (ilu-process:current-process))
       (prog1                         ; Yes, do A, releasing lock on way out.
         ...A...
         (ilu-process:process-unlock LOCK))
     ...B...                          ; No, do B, which does not depend on
lock.
     )
```

D.11 References

[Franz-92]: **Allegro Common Lisp User Guide**. Release 4.1. Berkeley, CA: Franz Incorporated, March 1992.

Bach, M.J., **The Design of the UNIX Operating System**. Englewood Cliffs, NJ: Prentice-Hall, 1986. Especially read Chapters 6, 7, and 8.

Deitel, H.M. **An Introduction to Operating Systems**. Reading, MA: Addison-Wesley, 1984. Especially read Part 2.

Kernighan, B.W. and R. Pike., **The UNIX Programming Environment**. Englewood Cliffs, NJ: Prentice-Hall, 1984. Especially read Sections 1.4 and 7.4.

Tanenbaum, A.S. **Operating Systems: Design and Implementation**. Englewood Cliffs, NJ: Prentice-Hall, 1987. Especially read Chapter 2.

Appendix E Porting ILU to Common Lisp Implementations

E.1 Introduction

The ILU runtime for Common Lisp is largely written in vanilla Common Lisp. The lisp-implementation-specific details are confined to a small number of macros and functions which need to be defined. (This assumes that you have a working port of ILU and its C support already on your operating system platform. If not, you will have to begin by doing that.) Aside from these macros and functions, you do not require anything not specified in the Common Lisp standard. You do not need Lisp code for TCP/IP or socket support. The major work is to write ilu-xxx.lisp, where `"xxx"` is the specifier for the particular implementation of Common Lisp in use, and any necessary xxx-to-C shims in ilu-xxx-skin.c. There are a number of things that have to be done in ilu-xxx.lisp. They can be regarded in three major sections: providing the ILU notion of foreign-function calls, connecting the Lisp's garbage collector to the ILU network GC, and providing either a threaded or event-loop model of operation. In addition, there is a small hook that has to be provided to convert between character sets.

E.2 Providing the ILU notion of foreign-function calls.

Perhaps the trickiest is to provide an implementation of the macro "define-c-function". This maps the ILU notion of a call into C into the native lisp notion. "define-c-function" has the signature

ilu::define-c-function (*LISP-NAME* `symbol`) (*DOC-STRING* `string`) (*C-NAME* `string`) (*ARGS* `list`) (*RETURN-TYPE* `keyword`) &key (*INLINE* `boolean cl:nil`) — Macro

The *LISP-NAME* is a symbol which will be the name of the function in Common Lisp. The *C-NAME* is a string which will be the ‘‘regular’’ C name of the C function to be called; that is, the name as it would be named in a C program, rather than the name of the symbol for the entry point of the function. *ARGS* is a list of arg which describe the signature of the C function, where each arg is either a keyword or a 2-tuple. If a keyword, the keyword indicates the type of the argument. Allowable argument types are

- `:short-cardinal` (unsigned-byte 16)
- `:cardinal` (unsigned-byte 32)
- `:short-integer` (signed-byte 16)
- `:integer` (signed-byte 32)
- `:short-real` (single-float)
- `:real` (double-float)
- `:byte` (0 <= fixnum < 256)
- `:boolean` (t or nil)
- `:fixnum` (-2^27 <fixnum < 2^27 (about))

- `:string` (string)
- `:constant-string` (string)
- `:bytes (vector of (unsigned-byte 8))`
- `:unicode` (Unicode if your Lisp supports it, vector of (unsigned-byte 16) otherwise)
- `:ilu-call` (unsigned-byte 32)
- `:ilu-object` (unsigned-byte 32)
- `:ilu-class` (unsigned-byte 32)
- `:ilu-server` (unsigned-byte 32)
- `:char*` (unsigned-byte 32)
- `:pointer` (unsigned-byte 32)

If the arg is a 2-tuple, the cadr is the type, and the car is the ''direction'', which may be either :in, `:out`, or `:inout`. Args with no ''direction'' are by default of direction `:in`. The *RETURN-TYPE* argument is a keyword for the return type of the function, which is drawn from the same set of keywords as the argument types. Return-types may also use the keyword :void, which specifies that no value is returned. The *INLINE* keyword is a boolean value which, if `cl:t`, indicates that the necessary type-checking has been assured by the application code, and that the C function may be called directly without type-checking the parameters.

`define-c-function` defines a Common Lisp function with a possibly different signature from the C function. This function has arguments which consist of all the `:in` and `:inout` arguments of the C function, in the order in which they occur in the signature of the C function. It returns possibly multiple values, which consist of the specified return type, if not `:void`, followed by any `:out` and `:inout` arguments to the C function, in the order in which they occur in the signature of the C function.

`define-c-function` assumes that the C function will call back into Common Lisp, and that gc may occur during the invocation of the C function. Therefore, any objects passed to C which are not values must be registered in some way to prevent them from moving during the call. Often this means that the actual call must be surrounded by code which makes static copies of, for example, strings, calls the C function, then frees the static copy after the call. In addition, when ''catching'' `:out` arguments and `:inout` arguments, it is usually necessary to pass a pointer to the appropriate argument, rather than the argument directly. Typically 1-element arrays have to be allocated to do this. The Franz ACL implementation uses a resource of arrays to minimize consing for this.

We should probably add another keyword, *NO-CALLBACKS*, to indicate that the C function will not call back into Common Lisp (and therefore some of the GC protection can be skipped when calling this function). Providing for *NO-CALLBACKS* in your implementation would probably be a good idea.

E.3 Network Garbage Collection

The Common Lisp-specific runtime must provide three calls which allow the kernel to map the kernel's C ILU object to a CLOS object. These are register-lisp-object, lookup-registered-lisp-object, and unregister-lisp-object. The idea behind these is to provide the C runtime with a handle on a CLOS object that is a small integer that will not be moved by Common Lisp GC, and to provide a layer which weak references can hide behind.

ilu::register-lisp-object (*OBJ* `ilu:ilu-object`) &key (*REFTYPE* `keyword` **:strong**) => `fixnum` Function

The OBJ is an ILU CLOS object (the Franz ACL implementation accepts any Common Lisp value except NIL, but this is only because it uses it internally in '`ilu-franz.lisp`'). The *REFTYPE* keyword may be either the keyword `:weak` or the keyword `:strong`, which determines whether the reference to the object is a weak reference or a strong reference. A weak reference is one that is not ''followed'' by the Common Lisp collector. The returned value is a fixnum that can be used with `lookup-registered-lisp-object` and `unregister-lisp-object` to find the object or remove the reference to the object, respectively.

ilu::lookup-registered-lisp-object (*INDEX* `fixnum`) => `ilu:ilu-object` Function

This function follows the reference indicated by *INDEX* and returns the object, or `cl:nil` if the *INDEX* is invalid.

ilu::unregister-lisp-object (*INDEX* `fixnum`) Function

Causes any reference indicated by *INDEX* to be removed.

ilu::optional-finalization-hook (*OBJ* `ilu:ilu-object`) Macro

This is a macro which should be defined in such a way as to indicate a finalization action for *OBJ* when the Common Lisp collector collects it. This finalization action will interact with the ILU kernel to ensure that remote peers of this Common Lisp will know that it no longer has an interest in the object. In addition, the finalization action will be able to prevent *OBJ* from being actually collected, should any peer have an active reference to it.

The Franz ACL implementation only allows the collector to run the finalization when it knows that no peer has a reference, by keeping the Common Lisp reference to the object as a strong reference until the C ILU kernel informs the Common Lisp ILU runtime that no peer has a reference, in which case the Common Lisp reference is changed to a weak reference. In time this allows the collector to GC the object, and the finalization action is called. The action that needs to be taken is ''null out'' both the pointer from the CLOS object to the C object, via `(setf (ilu-cached-kernel-obj lisp-obj) nil)`, and ''null out'' the reference from the C object to the CLOS object, via `(register-language-specific-object (kernel-obj lisp-obj) 0)`. See '`ilu-franz.lisp`', `ilu::franz-shutdown-ilu-object`, for an example. The Franz ACL example also does these shutdowns in a separate thread, instead of doing them directly in the GC finalization process. This is because the shutdown actions may cause arbitrary callbacks into Common Lisp, some of which may not occur on the stack of the ACL scheduler, which may invoke the collector.

If you feel that it just isn't possible to hook your Common Lisp collector into the network GC, you can simply define `register-lisp-object` to ignore the *REFTYPE* parameter, and define `optional-finalization-hook` to expand to nothing. The result will be that no ILU object in your address space will ever be GC'ed, and that no true instance of a collectible ILU object type referenced by your process will ever be GC'ed anywhere in its true address space until your Common Lisp image disappears. This might also be a good starting point, just to get the other parts working.

E.4 Thread and/or Event Loops

Every address space into which ILU is loaded is implicitly a server. This is partially because ILU uses method calls internally, such as pinging garbage collection callbacks, and partially because it provides for recursive protocols, in which a ''server'' might call back to a ''client'' during the execution of a method call. This means that any implementation of ILU has to provide a way to execute incoming calls; which means that it has to provide a stack and thread of control in which to execute the ''true'' code of the method call. There are two mechanisms supported by ILU to associate a thread of control with an incoming request, threads and event loops. In the thread model, each request is executed in a thread associated with either the specific request (thread-per-request) or the connection on which the thread arrives (thread-per-client). In the event loop model, one thread of control is multiplexed between all uses by means of calls into particular ''event handler'' routines when some ''event'' is delivered to the process. Typical events are timer expirations, I/O available on file descriptors, UNIX signals. Other more application-specific events are possible, such as X Window System events or XView toolkit events.

For a threaded Common Lisp, the thread model is preferred. To support this, the implementor of the Common Lisp runtime must call the C procedure `ilu_SetWaitTech()` with two C-callable routines that provide ways to block the current thread until input or output is available on a particular file descriptor. He must call `ilu_SetMainLoop()` with a main loop struct that provides NULL procedures for the `ml_run`, `ml_exit`, `ml_register_input`, and `ml_register_output` fields, simple procedures that return `ilu_FALSE` for the `ml_unregister_input` and `ml_unregister_output` fields, and three C-callable procedures that implement creation, setting, and unsetting of alarms for the `ml_create_alarm`, `ml_set_alarm`, and `ml_unset_alarm` fields. Finally, he must provide C-callable procedures to describe his thread system's mutex and condition variable system to the ILU C kernel, and register them by calling `ilu_SetLockTech()`. See the Franz ACL implementation for an example of this. Note that the file '`ilu-process.lisp`' provides an implementation-independent veneer over various process systems. It would be useful to extend that, then use it in providing the specific thread mechanisms, rather than using your Common Lisp's threads directly.

For an non-threaded Common Lisp, the event loop model is available. In this, you divide up all computation in your application into event handlers, separate functions that are run when some event occurs, and initialize the system by calling some event handler dispatcher routine, often called the ''main loop'' of the system. ILU provides a default main loop in the kernel, which provides support for two kinds of events:

timer expiration (ILU calls timers ‘‘alarms’’), and input or output available on a UNIX file descriptor. This means that handler functions can be registered to be called when an event of one of these types occurs. The ILU event loop is also ‘‘recursive’’; this means that event handlers can call back into the main loop to wait for something to occur. To use the ILU main loop, you must provide mainly a way to invoke the main loop, probably something like `ilu:xxx-main-loop`, where `"xxx"` is the name of your flavor of Common Lisp.

If the ILU main loop is for some reason not satisfactory, a Common Lisp-runtime-specific main loop can be substituted via a call to the ILU C kernel routine `ilu_SetMainLoop()`. This is often necessary to interoperate with UI toolkits like XView or Tk which believe that they own the main loop. Note that this main loop must provide all the functionality provided by the ILU main loop. A less-powerful main loop can be used *in addition to* the ILU main loop, by calling the ILU C kernel routine `ilu_AddRegisterersToDefault()`. See the comments in ‘*ILUSRC*`/runtime/kernel/iluxport.h`’ for documentation of all of this.

In addition to making the appropriate calls into the ILU kernel to set up either threaded mode or event-loop mode, the Common Lisp runtime implementor must provide a few required function calls:

ilu::initialize-locking Function

This misnamed function is called by the generic ILU Common Lisp runtime to set up the interaction mode, start the scheduler if necessary, and in general do anything necessary to initialize the Common Lisp-flavor-specific Common Lisp runtime.

ilu::setup-new-connection-handler (*FN* `function`) (*SERVER* `C-pointer`) (*PORT* `C-pointer`) Function

This is called when a client connects to a kernel server, *SERVER*, implemented in this address space. It should arrange to apply *FN* to `(list` *SERVER PORT*`)` if a new incoming connection is received on *PORT*. *FN* should return `cl:nil` if no handler could be established, non-`cl:nil` otherwise. *SERVER* is the C address of an ILU kernel `ilu_Server`, *PORT* is the C address of an ILU kernel `ilu_Port`. The ILU C kernel routine `ilu_FileDescriptorOfMooringOfPort()` will return the UNIX file descriptor of the `ilu_Mooring` of an `ilu_Port`. In threaded Common Lisps, this will typically cause a thread to be forked, which will watch for connections to this port. In event-loop Common Lisps, this will typically register *FN* as an event handler for `"`input available on the file descriptor of the mooring of *PORT*`"`.

ilu::setup-connection-watcher (*FN* `function`) (*CONN* `C-pointer`) (*SERVER* `C-pointer`) Function

This is called when a new connection is setup. It should arrange things so that *FN* is applied to `(list` *CONN SERVER*`)` whenever input is available on *CONN*. *FN* should return non-`cl:nil` if the input was successfully handled, `cl:nil` otherwise. If *FN* ever returns `cl:nil`, the connection-watcher should be demolished. *CONN* is the C address of an ILU kernel `ilu_Connection`, and *SERVER* is the C address of an ILU kernel `ilu_Server`. The ILU C kernel routine `ilu_FileDescriptorOfConnection()` will return the UNIX file descriptor for an `ilu_Connection`. In threaded Common Lisps, this will typically fork a thread which will handle requests coming in on this connection. In event-loop Common Lisps, this will typically register *FN* as an event handler for `"`input available on the file descriptor of the connection`"`.

E.5 Converting between character sets.

This section is not currently correct, but we are changing the Lisp runtime to make it correct.

ILU uses the ISO Latin-1 and Unicode (ISO 10646) character sets. Common Lisp uses a somewhat different version of 'character'. To provide for a mapping back and forth between ILU and Common Lisp, the runtime implementor must provide four macros:

ilu::construct-lisp-character-from-unicode (*UNICODE* (`unsigned-byte 16`))) => `character` Macro

ilu::determine-unicode-of-character (*LISP-CHAR* `character`) => `Unicode-code` Macro

ilu::construct-lisp-character-from-latin-1 (*LATIN-1-CODE* (`unsigned-byte 8`)) => `character` Macro

ilu::determine-latin-1-of-character (*LISP-CHAR* `character`) => `ISO-Latin-1-code` Macro

which I trust are self-explanatory.

E.6 Support for Dynamic Object Creation

ILU allows the dynamic creation of objects. This means that a true module can create the true CLOS object for an ILU object in a lazy manner, when it is referenced. The mechanism for doing this is called *object tables*. An object table consists of 2 C-callable functions, one to create an object, given its instance handle, and one to free any storage associated with the object table. To support this mechanism, the Common Lisp port of ILU has to provide the following function:

ilu::create-object-table (*OBJECT-OF-IH-FN* `function`) (*FREE-SELF-FN* `function`) => `C-pointer` Function

The function accepts two Lisp functions, and returns a pointer to a C struct of type `ilu_ObjectTable`, or the value `0`, if no object table pointer can be produced. The function will have to call into C space to actually produce the object table. Look at the Franz ACL implementation for an example of how to do this.

Appendix F Algorithm for Generation of Structural-Hash Type IDs

The following type ID construction algorithm is used for automatic generation of type IDs for all types and exceptions in ILU that have no explicitly specified type ID. The type ID is formed by (1) constructing a string containing all the salient information about the type or exception, in a standardized form, (2) taking the NIST Secure Hash Signature of that string, and (3) rendering the SHS in a string form as a 27-digit base-64 number.

F.1 Resolving Type Ambiguities

There are some types that can be described in multiple ways, using more or less generalized constructions. For example, the primitive integer types could also be described as fixed-point constructions. Every type that can be described multiple ways is always considered to *be* the one that uses the least general constructs. In particular, the primitive integer types are indeed considered primitive --- not fixed-point constructions.

F.2 Constructing the "salient information string"

This string consists of 8-bit bytes. Many of those bytes are described here using characters, which are taken to be in the US-ASCII character code.

<info string> ::= <self ref> <defn>*
<self ref> ::= <typeref> | <exnref>

The salient information string consists of a reference to the *initial item* (the type or exception being characterized), followed by the definitions of the interfaces, execeptions, and non-primitive non-explicitly-IDed types referenced directly or indirectly by the initial item. Each interface, exception, and non-primitive non-explicitly-IDed type that is referenced in the type string is defined exactly once in the type string, and these are the only definitions that appear in the type string. In particular, the string includes no definition of any primitive type, nor of any type whose ID is explicitly specified in the source language (explicit IDs can optionally be given in ISL by appending a `TYPEID` clause to a type definition; in OMG IDL there is a defined ID for every named type). The definitions appear in the order of first appearance of the corresponding reference. In other words, the definitions appear in breadth-first order. In particular, if the initial item has a definition (note that primitive types do not), then it is the first definition. Every definition of an item is preceded by at least one reference to that item. This forbids adding definitions of types in irrelevant cycles.

We could compress this a bit by making each definition immediately follow, and thus not duplicate, its first reference. But this gains little, and type fingerprints are not computed at runtime anyway. And it has this cost: it tempts an implementor to use a recursive strategy for generating the type string, and a recursive implementation will crash with a stack overflow when presented with hostile interface source code. The structure given is amenable to a simple worklist-based generation strategy.

F.3 References to Types, Interfaces, and Exceptions

A type reference is a reference to either a primitive type or a constructed one, or an explicit type UID (if the type UID has been explicitly specified).

```
<typeref> ::= <primitive> | <typeciteref> | <typeidref>
<typeciteref> ::= "(ref" <sp> <type cite> ")"
<typeidref> ::= "(id" <sp> <UID string> ")"
<UID string> ::= <string>
```

The primitive types are:

```
<primitive> ::=
  "byte"
| "shortcardinal"
| "cardinal"
| "longcardinal"
| "shortinteger"
| "integer"
| "longinteger"
| "shortreal"
| "real"
| "longreal"
| "shortcharacter"
| "character"
| "longcharacter"
| "boolean"
| "pickle"
| "void"
```

A reference to a constructed type cites two things: (1) the name of the interface in which the type is defined, and (2) the name given that type. These are ultimately <id>s, as defined in ISL; ISL identifiers may not contain any characters (e.g., quotes, parentheses) that are treated specially in the syntax of salient information strings. In a salient information string, <id>s are represented in US-ASCII (with no leading length nor trailing null).

```
<type cite> ::= <ifc ref> <sp> <typename>
<ifc ref> ::= <interfacename>
<interfacename> ::= <id>
<typename> ::= <id>
<sp> ::= a single space character (i.e., code 32(decimal))
```

Exception references similarly have two forms, depending on whether an ID has been explicitly specified:

```
<exnref> ::= "(exn" <sp> ( <exnciteref> | <exnidref> ) ")"
<exnidref> ::= "(id" <sp> <UID string> ")"
<exnciteref> ::= "(ref" <sp> <ifc ref> <sp> <exnname> ")"
<exn cite> ::= <ifc ref> <sp> <exnname>
<exnname> ::= <id>
```

F.4 Definitions of Types, Interfaces, and Exceptions

There are three kinds of things that need to be defined: types, interfaces, and exceptions.

<defn> ::= <type def> | <ifc def> | <exn def>

F.4.1 Types

A definition of a type gives its compound name, brand, and what the type is defined to be.

<type def> ::= "(type" <sp> <type cite> <sp> <brand> <sp> <type desc> ")"

A type can be defined either to be a primitive or a construction.

<type desc> ::= <primitive> | <construction>

Constructed types have different forms, specific to their types.

<construction> ::= <fixedpoint>
| <stringt>
| <array>
| <record>
| <optional>
| <sequence>
| <union>
| <enumeration>
| <object>

F.4.1.1 Alias Types

For alias types that have an explicitly assigned type ID, we simply generate a <typeidref> when it's referenced. For alias types that have the same type ID as their base type, we just use the base type in its place when constructing type info strings.

F.4.1.2 Array Types

This should be mostly obvious. An integer is given as its shortest decimal representation; positive integers have no sign character.

<array> ::= "(array" <sp> <typeref> (<sp> <dim>)+ ")"
<dim> ::= "(fixed" <sp> <non-negative integer> ")"
<integer> ::= "0" | (["-"] <non-zero-digit> <digit>*)
<digit> ::= <non-zero-digit> | "0"
<non-zero-digit> = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

F.4.1.3 Record Types

The field names are included on the grounds that they give clues to the semantics of the type, so different field names make different types.

<record> ::= "(record" (<sp> <rfield>)+ ")"

<rfield> ::= "(field" <sp> <fieldname> <sp> <typeref> ")"

<fieldname> ::= <id>

F.4.1.4 Optional Types

<optional> ::= "(optional" <sp> <typeref> ")"

F.4.1.5 Fixed-point Types

The fixed-point constructor gives the minimum and maximum numerators, and the fixed denominator value. Note that <integer> may be arbitrarily long.

<fixedpoint> ::= "(fixedpoint" <sp> <min-num> <sp> <max-num> <sp> <denom> ")"
<min-num> ::= <integer>
<max-num> ::= <integer>
<denom> ::= <positive integer> | "1/" <positive integer>

F.4.1.6 Sequence Types

<sequence> ::= "(sequence" <sp> <typeref> <sp> "(variable" <sp> <limit> "))"
<limit> ::= <integer>

There is always some finite limit, 2^32-1 when not explicitly given in ISL.

F.4.1.7 Union Types

<union> ::= "(union" <sp> <typeref> (<sp> <uarm>)+ ")"

<uarm> ::= "(arm" <sp> <typeref> <sp> ["(name" <arm name> ")" <sp>] "(" ["default"] ")"
(<sp> <discvalue>)* ")"

<arm name> ::= <id>

<discvalue> ::= "(val" <sp> (<string> | <integer> | ("TRUE" | "FALSE")) ")"

The <typeref> at the top level of a UNION constructor is the type of the tag, which must be an integer type, a character type, an enumerated type, or boolean. The <discvalue> is an integer when the tag is a numeric type, a string when the tag is an enumerated or character type, and a boolean literal when the tag is boolean.

F.4.1.8 Enumeration Types

```
<enumeration> ::= "(enumeration" (<sp> <efield>)+ ")"

<efield> ::= "(element" <sp> <id> <sp> <integer> ")"
```

Enumeration info includes the numeric codes assigned when not explicitly given.

F.4.1.9 Object Types

```
<object> ::= "(object"
 [ <sp> "(singleton" <sp> <singletoninfo> ")" ]
 [ <sp> "optional" ]
 [ <sp> "collectible" ]
 ( <sp> "(supertype" <sp> <typeref> ")" )*
 ( <sp> <method> )*

<method> ::= "(method" <sp> <methodname>
[ <sp> "asynchronous" ]
[ <sp> "functional" ]
<sp> "(returns" <sp> <typeref> (<sp> <exnref>)* ")"
( <sp> <marg> )*
     ")"

<marg> ::= "(parameter" <sp> <name> <sp> ( "in" | "out" | "inout" ) <sp> <typeref>
        [ <sp> "sibling" ] ")"
```

The form of object type constructions follows the features of the ISL fairly closely. A method with no explicit return result is considered here to return "void". The methods listed are only those introduced at the object type at hand --- the inherited methods are not listed.

```
<singletoninfo> ::= <string>
```

The <singletoninfo> is the string given in the ISL (quoted, as all <string>s are).

F.4.2 Interfaces

The definition of an interface simply gives its brand. We *could* make it also include a reference to everything defined in that interface, but I don’t see any particular value in doing so.

```
<ifc def> ::= "(interface" <sp> <interfacename> <sp> <interfacebrand> ")"
<interfacebrand> ::= <string>
```

F.4.3 Exceptions

The definition of an exception gives its compound name, its brand, and the associated datatype. As exceptions are not yet branded in the rest of ILU, the brand here is always the empty string. If the exception has no associated datatype, it is given as "void" here.

<exn def> ::= "(exception" <sp> <exn cite> <sp> <exnbrand> <sp> <typeref> ")"
<exnbrand> ::= <string>

F.4.4 Brands

<brand> ::= <string>

The string brand for a type, exception, or interface. If not explicitly given in the ISL, the empty string.

F.4.5 String Literals

A <string> is encoded using US-ASCII and surrounded with double-quotes (US-ASCII code 34). If the double-quote character appears in the string, it is escaped with a leading backslash character. If a backslash character appears in the string, it is escaped with an additional leading backslash character. If a non-printing character (with a US-ASCII code less than 32(decimal) or greater than 126(decimal)) appears in the string, it appears as a 3-digit octal number containing the US-ASCII code for that character, preceded with a backslash character.

<string> ::= <dquote> <string-char>* <dquote>
<dquote> ::= the US-ASCII double-quote character (i.e., code 34(decimal))
<string-char> ::= <simple-char> | <quoted-char>
<simple-char> ::= any printing US-ASCII character (i.e., codes 32(decimal) to 126(decimal)),
except double-quote (code 34(decimal)) or backslash (code 92(decimal))
<quoted-char> ::= <quoted-backslash> | <quoted-double-quote> | <quoted-byte>
<quoted-backslash> ::= <backslash> <backslash>
<backslash> ::= the US-ASCII backslash character (i.e., code 92(decimal))
<quoted-double-quote> ::= <backslash> <dquote>
<quoted-byte> ::= <backslash> <octal-digit> <octal-digit> <octal-digit>
<octal-digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"

F.4.6 Example

Consider the following interface:

```
INTERFACE Foo BRAND "13.2.116.14 October 26, 1998";

EXCEPTION E;

TYPE C1 = FIXEDPOINT
MIN-NUMERATOR 0
MAX-NUMERATOR 59
        DENOMINATOR 1;
```

```
TYPE I1 = FIXEDPOINT
MIN-NUMERATOR -2147483648
MAX-NUMERATOR 2147483647
        DENOMINATOR 1 ;

TYPE O-1 = OBJECT
METHODS m1(o:O-2, c:C1):I1 END;

TYPE O-2 = OBJECT
SUPERTYPES O-1 END
METHODS m2() RAISES E END END
BRAND "13.2.116.14 October 26, 1998" ;

TYPE O-1X = O-1;

TYPE O-1Y = O-1X TYPEID "xyz:bad-idea";
```

The salient information strings for types O-1 and O-1X are the same. That string is:

```
(ref Foo O-1)
(interface Foo "13.2.116.14 October 26, 1998")
(type Foo O-1 "" (object (method m1 (returns integer) (parameter o in (ref
Foo O-2)
) (parameter c in (ref Foo C1)))))
(type Foo O-2 "13.2.116.14 October 26, 1998" (object (supertype (ref Foo
O-1)) (
method m2 (returns void (exn (ref Foo E))))))
(type Foo C1 "" (fixedpoint 0 59 1))
(exception Foo E "" void)
```

Line breaks have been introduced into the display above for readability; the actual salient information string contains none of those linebreaks nor any whitespace of any kind at their positions.

The type ID (not salient information string) for O-1Y is `"xyz:bad-idea"`.

F.5 Calculation of the NIST SHS of the "salient information" String.

See FIPS PUB 180-1: Secure Hash Standard (`http://www.itl.nist.gov/div897/pubs/fip180-1.htm`).

Abstract:
‘‘This standard specifies a Secure Hash Algorithm (SHA-1) which can be used to generate a condensed representation of a message called a message digest. The SHA-1 is required for use with the Digital Signature Algorithm (DSA) as specified in the Digital Signature Standard (DSS) and whenever a secure hash algorithm is required for Federal applications. The SHA-1 is used by both the transmitter and intended receiver of a message in computing and verifying a digital signature.’’

F.6 Conversion of the SHS to a Base-64 Number.

The SHS is converted to a string of length 32 bytes, each containing an ASCII character code. First, two 0 bits are appended to the 160-bit SHS. This 162-bit string is encoded as 27 6-bit digits, most significant digit first. The encoding uses characters 'a' - 'z' to represent the values 0-25, the characters 'A' - 'Z' to represent the characters 26-51, the characters '0' - '9' to represent the values 52-61, the character '-' to represent the value 62, and the character '+' to represent the value 63. Finally the 5 bytes `"ilut:"` are prepended, giving the total of 32 bytes.

Appendix G Possible ISL Name Mappings for Target Languages

This note outlines a proposal for name mappings and restrictions; this proposal is not yet accepted. (Thanks to external standards such as CORBA, this proposal cannot be implemented for some languages, such as ANSI C.) **The mappings outlined here are not necessarily the ones used in the current ILU release.**

This proposal is about how to name things in the various programming languages, in a way that avoids name clashes. It imposes no restrictions on the ISL source. However, the mappings will be more straightforward if the ISL source avoids two things: (1) two or more concsecutive hyphens in a name, and (2) starting an interface or type name with ‘‘ilu-’’ (in any casing).

The first step in mapping an ISL to a programming language is to scan type and interface names for the substring ‘‘ilu-’’ (in any casing); wherever it occurs, we insert a trailing digit zero.

In a similar way, we next scan the name for sequences of hyphens. Wherever two or more hyphens appear consecutively, the digit zero (‘0’) is inserted after every other one, starting with inserting a zero after the second hyphen.

The following steps assume the first two steps have already been done.

Where tuples <N1, N2, ... Nk> of ISL names must be mapped into a flat programming namespace, we concatenate the ISL names, with a double hyphen (‘‘--’’) inserted between each.

Where ISL names (or tuples thereof) must be mapped, together with ILU-chosen names derived from the ISL names, into a flat programming namespace, the derived names begin with fixed strings specific to the derivation, where the fixed strings begin with ‘‘ilu-’’ (with any case), and a double hyphen is inserted between the fixed string and the ISL name.

Where ISL names (or tuples thereof), and possibly ILU-chosen names derived from the ISL names, must be mapped, together with a fixed set of ILU-chosen names, into a flat programming namespace, the fixed ILU-chosen names begin with ‘‘ilu-’’ (with any case) and do not include a double hyphen.

The final step is to translate hyphens to underscores, for programming languages that accept underscores but not hyphens in names.

Following is a specification of how names are mapped in each language. The notation "[N]" is used to denote the application of the first two steps and the last step. Examples of "[..]" are:

```
[Foo] => Foo
[foo-bar] => foo-bar
[wait----for---it-] => wait--0--0for--0-it-
[iluminate] => iluminate
[ilu---uli] => ilu-0--0uli
```

The mappings also use the notation "[[..]]" to denote the mapping of a type-reference.

G.1 C mapping

[This mapping, while clean, will never be adopted because of the more problematic mapping specified by the OMG's CORBA document.]

Item `N` from interface `I` is mapped to `[I]__[N]`. `[[I.N]] = [I]__[N]`; `[[N]] = [I]__[N]`, where `I` is the current interface.

An enumerated value named `V`, of type `T` in interface `I` is mapped to `[I]__[T]__[V]`.

A declaration of a record type `T` in interface `I` with fields `F1:TR1`, ... `Fn:TRn` is mapped to

```
typedef struct {[[TR1]] F1; ... [[TRn]] Fn} [I]__[T];
```

A declaration of a union type `T` in interface `I` of types `TR1`, ... `TRn` is mapped to

```
typedef enum {[[I.T]]__[[TR1]], ... [[I.T]]__[[TRn]]} ilu_tags__[[I.T]];
typedef struct {ilu_tags__[[I.T]] tag;
    union {
        [[TR1]] [[TR1]];
        ...
        [[TRn]] [[TRn]];
    } val;
} [[I.T]];
```

For passing exceptions through the method calls in interface `I`, the following auxiliary declaration is generated (supposing exceptions `ER1:TR1`, ... `ER2:TR2` are raised):

```
typedef struct {
    ilu_Exception returnCode;
    union {
        [[TR1]] [[ER1]];
        ...
        [[TRn]] [[ERn]];
    } val;
} ilu_Status__[I];
```

An object type named `T` in interface `I` with methods `M1`, ... `Mn` maps to

```
typedef ilu_Ojbect [[I.T]];
[result-type-1] [I]__[T]__[M1]([[I.T]] ilu_self,
    [[arg-type-1-1]] [arg-name-1-1], ...
    [[arg-type-1-k]] [arg-name-1-k]);
...
```

G.2 C++ mapping

Item `N` from interface `I` is mapped to `[I]__[N]`. `[[I.N]] = [I]__[N]`; `[[N]] = [I]__[N]`, where `I` is the current interface.

A declaration of an enumerated type named `T` in interface `I` containing values `V1`, ... `Vn` is mapped to `typedef enum {[V1], ... [Vn]} [I]__[T]`.

Record and union declarations are mapped as for C. The exception status declaration is as for C.

G.3 Modula-3 mapping

G.4 Mapping ILU ISL to Modula-3

Version 1 of ILU supported Modula-3, and this section describes the mapping we worked out for it.

G.4.1 Names

An item named `Bar` in ISL interface `Foo` becomes an item named `Bar` in the Modula-3 interface `Foo`. A hyphen in an ISL name becomes an underscore in the corresponding Modula-3 name.

G.4.2 Types

ISL types appear in Modula-3 as follows:

1. `SHORT INTEGER` becomes `[-32768 .. 32767]`.
2. `INTEGER` becomes `INTEGER`.
3. `LONG INTEGER` becomes

```
TYPE LongInt = RECORD
                 high: [-16_80000000 .. 2147483647];
                 low : Word.T (*[0 ..  4294967295]*)
               END;
```

   This represents the number `high*2^32 + low`. We always have the invariants `-2^31 <= high < 2^31` and `0 <= low < 2^32`, even on systems whose natural word size is greater than 32 bits.
4. `BYTE` becomes `[0 .. 255]`.
5. `SHORT CARDINAL` becomes `[0 .. 65535]`.
6. `CARDINAL` becomes `Word.T`.
7. `LONG CARDINAL` becomes `RECORD high, low: Word.T END`. This representation works analogously to that for `LONG CARDINAL`.
8. `SHORT REAL` becomes `REAL`.
9. `REAL` becomes `LONGREAL`.
10. `LONG REAL` becomes an opaque type. Values of this type can only be handed around; no other operations are provided, not even equality testing. `LONG REAL` is not really supported yet.
11. `SHORT CHARACTER` becomes `['\000' .. '\377']`.
12. `CHARACTER` becomes `[0 .. 65535]`.

13. Variable-length `ARRAY`s of `SHORT CHARACTER` become `TEXT`.
14. Other variable-length arrays become `REF ARRAY OF`.
15. Fixed-length arrays of `SHORT CHARACTER` become arrays of `BITS 8 FOR ['\000' .. '\377']`.
16. Fixed or variable-length `ARRAY`s of `BYTE` become arrays of `BITS 8 FOR [0 .. 255]`.
17. No other arrays specify packing in the Modula-3.
18. A fixed length array, `ARRAY OF` *L1*`, ...` *Ln*, becomes `ARRAY [0 ..` *L1*`-1] OF ... ARRAY [0 ..` *Ln*`-1] OF`.
19. An ISL record becomes a M3 record.
20. An ISL union becomes a M3 object type and some subtypes. The ISL

```
TYPE Foo = DiscT UNION
    case1: T1 = val1-1, ... val1-j END,
    ...
    casen: Tn = valn-1, ... valn-k END
    END OTHERS;
```

   maps to the Modula-3

```
TYPE Foo = BRANDED OBJECT d: DiscT END;
TYPE  Foo_case1 = Foo BRANDED OBJECT v: T1 END;
CONST Foo_case1__Code : DiscT = val1-1;
...
TYPE  Foo_casen = Foo BRANDED OBJECT v: Tn END;
CONST Foo_casen__Code : DiscT = valn-1;
TYPE  Foo_OTHERS = Foo BRANDED OBJECT END;
(* Where every Foo is of one of the subtypes enumerated here,
   and the tag field (d) is consistent with the subtype. *)
```

   The *Foo*`_OTHERS` subtype appears only for union constructions including the `OTHERS` keyword. If the ISL union has a `DEFAULT` arm

```
    cased: Td = DEFAULT
```

   it maps to another subtype in Modula-3:

```
TYPE  Foo_cased = Foo BRANDED OBJECT v: Td END;
```

   The *Foo_casen*`__Code` constants are conveniences for filling in and decoding the `d` field. Note that code that creates a *Foo* is responsible for filling in the `d` field.

21. An ISL enumeration becomes a M3 enumeration. Due to the fact that Modula-3 offers no way to specify the codes used to represent enumerated values, the codes specified in ISL, if any, have no effect on the translation.
22. When a *Foo* becomes a *Bar*, an `OPTIONAL` *Foo* becomes a `REF` *Bar*, unless *Bar* is a subtype of `REFANY`, in which case `OPTIONAL` *Foo* becomes *Bar*; `NIL` encodes the `NULL` case.
23. An ISL object type becomes a Modula-3 object type. The ISL adjectives `SINGLETON`, `DOCUMENTATION`, `COLLECTIBLE`, `OPTIONAL`, `AUTHENTICATION`, and `BRAND` have no effect on the mapping into the Modula-3 type system.

`OUT` and `INOUT` method parameters in ISL become `VAR` parameters in Modula-3; `IN` parameters become `VALUE` (by default) parameters. The `SIBLING` constraint in ISL has no manifestation in the Modula-3 type system.

The methods are declared to raise the exceptions `IluBasics.Failed` and `Thread.Alerted` in addition to the exceptions declared in the ISL. Exception `IluBasics.Failed` is used to convey all the errors that can arise from the RPC mechanism, except `Thread.Alerted`. Is the surrogate (and the other surrogates from the same server?) broken after either of these exceptions is raised?

Because ILU has multiple inheritance (i.e., an object type can have more than one direct supertype), the Modula-3 subtype relation is a sub-relation of the ILU subtype relation. In general, an ILU object type is mapped to a suite of Modula-3 object types, and a cohort of Modula-3 objects (one of each of the suite of Modula-3 types) correspond to one ILU object. There will be only one Modula-3 object (type) when only single-inheritance is used in constructing the ILU object type: when every ancestor type has at most one direct ancestor. Except where the programmer knows this is the case, and plans for it to remain so, she must abandon the native Modula-3 `TYPECASE`/`NARROW`/automatic-widen facilities for explicit calls that invoke the ILU subtype relation.

To generalize the Modula-3 `TYPECASE`/`NARROW`/automatic-widen facilities, the Modula-3 object type `Ilu.Object` includes the following method:

```
PROCEDURE ILU_Qua_Type(ot: ObjectType): Object;
```

If the object has, in ILU, the given object type, the Modula-3 object of the appropriate Modula-3 type is returned; otherwise, NIL is returned. As an added convenience, the Modula-3 mapping of interface *Foo* will contain, for each of its object types *Bar*:

```
PROCEDURE ILU_Qua_Bar(x: Ilu.Object): Bar;
```

This procedure takes a non-`NIL` argument. If the argument is, in ILU, an instance of *Bar* or one of its subtypes, the corresponding language-specific object is returned; otherwise, `NIL` is returned.

G.4.3 Exceptions

ISL exceptions are exactly like Modula-3 exceptions, and are mapped directly.

G.4.4 Example

Here's a sample ISL spec, and the resulting Modula-3 mappings:

```
INTERFACE Foo;

TYPE String = ilu.CString;
TYPE UInt = CARDINAL;

TYPE E1 = ENUMERATION val1, val2, val3 = 40 END;
TYPE R1 = RECORD field1 : CARDINAL, field2 : E1 END;
TYPE FAB = ARRAY OF 200 BYTE;
TYPE VAB = SEQUENCE OF BYTE;
```

```
TYPE FASC = ARRAY OF 10 SHORT CHARACTER;
TYPE VASC = SEQUENCE OF SHORT CHARACTER;
TYPE FAC = ARRAY OF 5 CHARACTER;
TYPE VAC = SEQUENCE OF CHARACTER;
TYPE A2 = ARRAY OF 41, 3 R1;
TYPE S1 = SEQUENCE OF E1;
TYPE U1 = UNION R1, A2 END;

EXCEPTION Except1 : String;

CONSTANT Zero : CARDINAL = 0;

TYPE O1 = OBJECT
    METHODS
        M1(r1: R1, INOUT v: VASC, OUT s1: S1): UInt RAISES Except1 END,
        FUNCTIONAL Hash(v: VASC): FASC,
        ASYNCHRONOUS Note(x: LONG REAL)
    END;
```

The Modula-3 mapping:

```
INTERFACE Foo;

IMPORT Ilu, IluBasics, Thread;
IMPORT ilu; <*NOWARN*>

TYPE UInt = CARDINAL;
TYPE E1 = {
  val1,
  val2,
  val3};
TYPE R1 = RECORD
   field1 : CARDINAL;
   field2 : E1;
END;
TYPE VASC = TEXT;  (* NIL not allowed *)
TYPE S1 = REF ARRAY OF E1;  (* NIL not allowed *)
TYPE FASC = ARRAY [0..9] OF Ilu.PackedShortChar;


(* declaration of M3 type "Foo.O1" from ILU class "Foo:O1"  *)

TYPE O1 = Ilu.Object OBJECT
  METHODS
    M1 (r1: R1; VAR v: VASC; VAR s1: S1): UInt
       RAISES {IluBasics.Failed, Thread.Alerted, Except1};
    Hash (v: VASC): FASC RAISES {IluBasics.Failed, Thread.Alerted};
    Note (x: Ilu.LongReal) RAISES {IluBasics.Failed, Thread.Alerted};
  OVERRIDES
    ILU_Get_Type := ILU_Get_Type_O1
```

```
  END;

PROCEDURE ILU_SBH_To_O1 (sbh: TEXT; mostSpecificTypeID: TEXT := NIL): O1
  RAISES {IluBasics.Failed, Thread.Alerted};

PROCEDURE ILU_Get_Type_O1 (self : Ilu.Object): Ilu.ObjectType;

PROCEDURE ILU_Qua_O1 (x: Ilu.Object): O1;

TYPE A2 = ARRAY [0..40] OF ARRAY [0..2] OF R1;
TYPE  U1 = BRANDED OBJECT d: Ilu.ShortInt END;  (* NIL not allowed *)
TYPE  U1_R1       = U1 BRANDED OBJECT v: R1 END;
CONST U1_R1__Code : [-32768..32767] = 0;
TYPE  U1_A2       = U1 BRANDED OBJECT v: A2 END;
CONST U1_A2__Code : [-32768..32767] = 1;
TYPE VAC = REF ARRAY OF Ilu.Character;  (* NIL not allowed *)
TYPE FAC = ARRAY [0..4] OF Ilu.Character;
TYPE VAB = REF ARRAY OF BITS 8 FOR Ilu.Byte;  (* NIL not allowed *)
TYPE FAB = ARRAY [0..199] OF Ilu.PackedByte;
TYPE String = TEXT;  (* NIL not allowed *)

CONST Zero : CARDINAL = 0;

(* Exceptions *)

EXCEPTION Except1 (String);

END Foo.
```

Index of Concepts

Index of Functions, Variables, Methods, and Types

This Page Intentionally Left ‘‘Blank’’.

Table of Contents