Using ILU with Standard C

Bill Janssen <janssen@parc.xerox.com>

(Typeset 11 July 1997)

1 ILU with Standard C

1.1 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

1.2 The ISL Mapping to C

1.2.1 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`.

Enumeration value names are formed by prepending the interface name and "_" to the ISL enumeration value name. Enumeration names and values are then cast into C `enum` statements.

Constant names are prepended with their interface name. They are implemented with the `const` declaration statements.

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, hyphens are replaced with underscores.

1.2.2 Mapping Type Constructs Into C

1.2.2.1 Basic Types

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

- `BOOLEAN => CORBA_boolean`
- `BYTE => CORBA_octet`
- `CHARACTER => ilu_character`
- `SHORT CHARACTER => CORBA_char`
- `CARDINAL => CORBA_unsigned_long`
- `SHORT CARDINAL => CORBA_unsigned_short`
- `LONG CARDINAL => ilu_longcardinal`
- `INTEGER => CORBA_long`
- `SHORT INTEGER => CORBA_short`
- `LONG INTEGER => ilu_longinteger`
- `REAL => CORBA_double`
- `SHORT REAL => CORBA_float`
- `LONG REAL => ilu_longreal`

1.2.2.2 Records

Records map directly into corresponding C structures.

1.2.2.3 Unions

Because of the somewhat baroque CORBA specification, 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

```
typedef struct _I_u1_union I_u1;
enum I_u1_allowableTypes {
  I_u1_integer,
  I_u1_shortreal
};
struct _I_u1_union {
  enum I_u1_allowableTypes _d;
  union {
    ilu_integer integer;
    ilu_shortreal shortreal;
  } _u;
};
```

Note the discriminator `_d` may take on the values of `I_u1_integer` or `u_u1_shortreal` indicating how to interpret the data in the union. Also note how the enumerated names are formed: with the interface name and the type name prepended to the enumeration element name.

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
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 {
    ilu_integer a;
    ilu_shortreal b;
```

```
      ilu_real c;
    } _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 integer. When the discriminator has the value `I_green` the member `b` has a valid value and the type is interpreted to be shortreal. If the discriminator has any other value, the member `c` is expected to have a valid value and the type is interpreted to be ilu_real (double).

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

1.2.2.4 Floating Point Values

The ISL `SHORT REAL` primitive type maps to the C `float` data type while `REAL` maps to `double`. The ISL `LONG REAL` primitive type currently doesn't map to anything real.

1.2.2.5 Sequences

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;
  long *_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);
    ...
```

1.2.2.6 Pickles (CORBA 'any')

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(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_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_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.

1.2.3 Objects and Methods

1.2.3.1 Basics

ILU uses an object system embedded into C. The type of objects in this system is `ILU_C_Object*`; the identifiers can `ILU_C_OBJECT` and `CORBA_Object` can also be used for this type.

`ILU_C_OBJECT` is a macro that expands to the identifier for the C object type specified by the version of CORBA being used (currently 2.0).

C has no subtyping relationship isomorphic to that of ISL object types, so 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-specifc C type name may be used to express intent. For conformance with CORBA, the C name for ISL type *I.T* is *I*_*T*.

Classes are reified, as C type `ILU_C_Class`. To create a true object, an application must first create an `ILU_C_Class`, and then instantiate that class (surrogate objects are created by stubs, which use private stub-generated `ILU_C_Class`es). 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 suite 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 class. 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*.

The C mapping of an object type includes a suite 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 (`CORBA_Object`) that should have type *I.T* (among possibly others).

A *binding procedure* is a procedure that takes some name for an object instance, and returns the actual instance. Users of a module typically use a surrogate-side binding procedure, which takes the string binding handle of the object, and the most specific type ID of the object's type (if known). Suppliers of a module typically bind objects with a creation procedure, which takes an instance ID, a server on which to maintain the object, and arbitrary user data, and creates and returns the true instance of the object. A generic creation procedure, `ILU_C_CreateTrueObject`, is declared in '`iluchdrs.h`' for application use. Also, the server-side stub for *I* offers a pair of creation procedures for each object type *I.T*; these creation procedures instantiate the default class for *I.T*.

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

`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.

`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 1.4.4.1 [Using C Object Tables], page 22). 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.

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.

Finally, 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;
  int xx;
  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);
  }
  xx = 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", xx );
  return(0);
}
```

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

In this example, the string binding handle is obtained from standard input along with some floating-point value.

The class specific function I_T__CreateFromSBH is 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 ilu_real argument f. I_T_M returns an ilu_integer value which is placed in xx.

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.

1.2.3.2 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.

1.2.3.3 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.

1.2.4 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 );
        }
      ...
```

1.2.5 Parameter Passing Considerations

Here is ILU's version of table 20 from the CORBA 2.0 spec.

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*

Here `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.

1.2.6 True Module Construction

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 1.2.3.2 [Interface Inheritance], page 10. 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 ()
{
  calc_self s;
  char * sbh;
  CORBA_Environment ev;

  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 (ILU_C_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));
          exit(1);
        }
    }
}
```

1.2.7 Using ILU Modules

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.

1.2.8 Stub Generation

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 interface foo to ./foo.h...
code for interface foo to ./foo-common.c...
code for interface foo to ./foo-surrogate.c...
code for server stubs of interface foo to ./foo-true.c...
%
```

1.2.9 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.

1.3 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.

1.4 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).

1.4.1 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.

1.4.2 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.

`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** (Function
RETAIN(char *) *sid*, RETAIN(char *) *ih*, ilu_Class *static-class*)

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)) Function
ILU_C_CreateSurrogateObject (ilu_Class *type*, RETAIN(char *) *ih*, ilu_Server *server*, ILU_C_ENVIRONMENT **env*)

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.

1.4.3 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 Function
ILU_C_CopyIdentity(RETAIN(ilu_IdentityInfo) 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 See ⟨undefined⟩ [Security], page ⟨undefined⟩ and the file '*ILUSRC*`/runtime/kernel/iluxport.h`' for more information on identities and identity schemes.

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

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

1.4.4 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. If the main thread cannot be terminated until the program ends, the call will block appropriately.

`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_Server)` **ILU_C_InitializeServer** (`OPTIONAL(RETAIN(char *))` *serverID*, `OPTIONAL(GLOBAL(ILU_C_ObjectTable))` *obj_tab*, `OPTIONAL(RETAIN(char *))` *protocol*, `OPTIONAL(RETAIN(ilu_TransportInfo))` *transport*, `OPTIONAL(RETAIN(ilu_Passport))` *identity*, `ilu_boolean` *setdefaultport*) — Function

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 *setdefaultport*, 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 `Sun RPC`. *transport*, if not `NULL`, is a sequence of strings that specifies the transport stack to use below the RPC protocol; `NULL` signifies use of SunRPC Record Marking over TCP to/from one of the IP addresses of this host. Chapter 8 gives details on protocol and transport strings. If an *identity* is specified, it may be used for communications security purposes. If *setdefaultport* is true, the newly created `ilu_Port` will become the default port of the `ilu_Server`.

1.4.4.1 Using C 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*. A value of type `ILU_C_ObjectTable` may be created by a call on

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

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.

1.4.5 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*`)`.

1.5 Index

C

I

O

S

B

C