Using ELROS To Implement ISO Application Protocols

M.L. Branstetter
J.A. Guse
D.M. Nessett
L.C. Stanberry

"*Elros* Son of Eärendil and Elwing, who at the end of the First Age chose to be numbered among Men, and became the first King of Númenor (called *Tar-Minyatur*), living to a very great age." [**The Silmarillion**, J.R.R. Tolkien]

DISCLAIMER

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Acknowledgments

The design and implementation of a sophisticated system like ELROS generally involves a large number of contributors. We greatly appreciate and acknowledge the work and assistance of the following people without which we could not have brought ELROS into the world.

Robyne Sumpter worked with us for approximately a year on the implementation of the ELROS runtime system. She provided us with very valuable expertise in multi-tasking and communications interfaces. Robyne left the ELROS project to become a project leader in the storage group and we wish her all the best in her new responsibilities.

Rob Shaw spent about five months reworking the ELROS preprocessor parser, converting it to an enhanced C-based grammar. We didn't have much time to get to know Rob, but during his time with the project, we were very impressed with his abilities. We are sure that he will excel in his graduate studies in computer science and will miss his exuberance and strong work ethic.

Both Gary Long and Ashley Allen were early participants in the ELROS design effort. While their tenure with the project was short, several features of the ELROS embedded language are the result of their suggestions and contributions.

Mark Boolootian has recently joined the ELROS project to work on optimizing communications performance. We expect him to make major contributions to our work in the future.

Notice, Disclaimer, and Conditions of Use

I. Introduction

Developing distributed applications is a sometimes onerous chore rarely made easier by good programming tools. ELROS is an attempt to improve this situation, at least with respect to writing distributed application code.

ELROS stands for **E**mbedded **L**anguage for **R**emote **O**perations **S**ervice. Remote Operations Service is a generalization of remote procedure call (RPC), being different by virtue of its potential asynchronicity. The model and protocol associated with remote operations are ISO standards (ISO 9072-1 and 9072-2), which uses the services of other ISO standard protocols, including Abstract Syntax Notation 1, abbreviated ASN.1, (ISO 8824) and the ISO presentation-layer protocol (ISO 8822 and 8823). ELROS provides specification and runtime interfaces to control certain aspects of each of these protocols.

The designers of ELROS intend it to be used for a variety of purposes. The main goal is to provide a convenient programming language for writing distributed scientific applications. This affected the design mainly by motivating certain language primitives, such as the call statement, elidable parameters, and the runtime specification of the length of Arrays that are operation parameters. A second goal was to support the writing of programs that implement ISO application layer protocols.

An ELROS primer is available[1] that explains the use of ELROS in writing distributed scientific applications. The purpose of this tutorial is to explain the features of ELROS that are useful in writing implementations of ISO application protocols.

It is assumed that readers of this document are generally familiar with the ISO remote operations model, ASN.1 and the services of the ISO presentation layer. Readers unfamiliar with these concepts should consult the appropriate literature before proceeding[2]. Also readers should be

---

[1] M.L. Branstetter, J.A. Guse, D.M. Nessett and L.C. Stanberry, "An ELROS Primer", Lawrence Livermore National Laboratory Technical Report UCRL-MA-108543 (rev. 1), Oct. 16, 1992.

[2] A good introduction to ISO protocols, including the ISO presentation layer, ASN.1 and remote operations is found in The Open Book by Marshall Rose, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

familiar with the concepts of distributed programming, including the client/server model (or to use the corresponding ISO terminology, the invoker/performer model); communications issues such as timeouts, addresses, and the various types of communications failures; task/thread models of concurrent computing and presentation layer functions to accommodate big-endian/little-endian incompatibilities, different numerical representations (e.g., different floating point formats), and different representations of complex data structures such as Records, Arrays and Unions. Since ELROS statements are embedded in C language programs, readers should be familiar with that language. Finally, to promote pedagogical clarity this tutorial assumes ELROS runs over the ISODE communications platform; although, ELROS is designed to be communications platform independent. Readers familiar with ISODE will find some parts of the document easier to comprehend than those unfamiliar with that platform.

II. Writing Simple ELROS Programs

To ensure the reader shares with us a common set of concepts to describe the advanced features of ELROS, we provide a basic outline of how to write elementary ELROS programs. Readers who find this description difficult are advised to read the ELROS primer[3] before proceeding.

2.1 The ELROS Model of Programming

A distributed application is composed of many parts, which we call its *components*. Distributed application components normally interact with each other in pairs. During a particular interaction, one component requests service from another. We say that the component seeking service acts in the role of an *invoker*, while the component providing the remote service acts in the role of a *performer* (fig. 1).

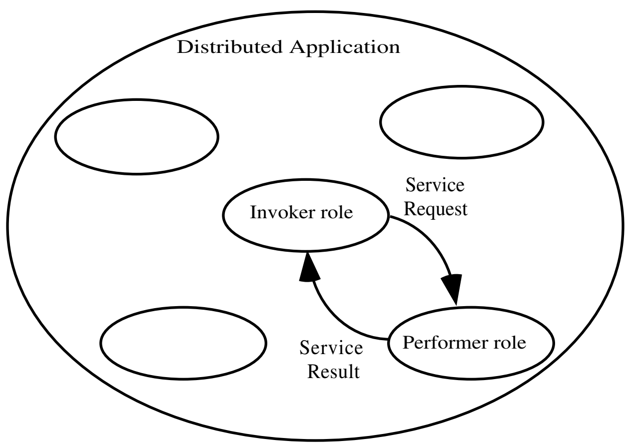

Figure 1 - Distributed application components acting in the role of invoker and performer

In general components are not limited to one role. They may act as an invoker when interacting with one component and act as a performer while interacting with another. It is even common for components to act in the

---

[3] See footnote 1.

role of an invoker and concurrently act in the role of a performer for another interaction (fig. 2).

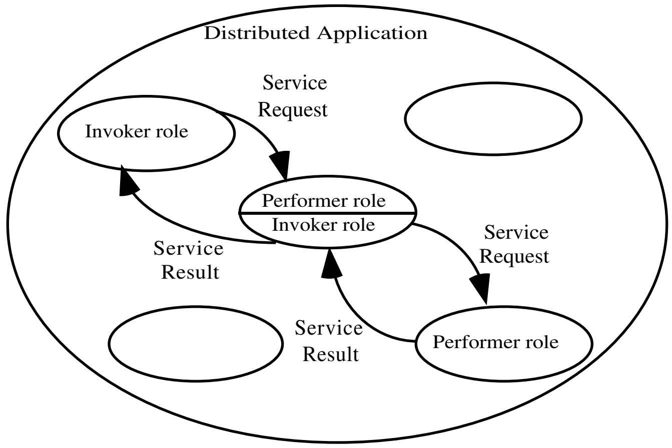

Figure 2 - Distributed application component concurrently acting as both invoker and performer

When the role of a remote component can be either an invoker or performer, we refer to it as a *correspondent*.

The actions that a component takes when assuming the role of an invoker are different from its actions when assuming the role of a performer. As an invoker, a component must locate the performer that supplies the remote service, make a service request of it and then accept the results (if any) that it returns. A performer waits for a service request, carries out the activity that satisfies the request and then returns any results of that activity to the invoker.

Invokers and performers can interact in two different ways. If the invoker makes a service request and then blocks its activity until the performer returns the results, we say the interaction is *synchronous*. If the invoker makes a request and then continues computing while the request is being processed by the performer, waiting at some later point in time for the results of the request, we say the interaction is *asynchronous*. Generally, synchronous interactions are useful when the invoker cannot continue computing until the results of a service request are received. On the other hand, asynchronous interactions are useful in two cases. There are situations in which an invoker wishes to compute in parallel with the performer. This requires it to make asynchronous service requests so that it does not block. Secondly, sometimes an invoker desires to make several requests of a performer in a closely connected series, not blocking on each individual request. Sending several requests to a performer before receiving results is known as *batching*.

A service request made by an invoker will generally contain a parameter that specifies data required by the performer to satisfy the request. Obviously, the invoker and performer must agree on the identity and data types of operation parameters so that their interactions are not confusing. Definition of the services offered by a performer (and used by an invoker), including the definition of parameters for service requests, is collected in one place called an *interface*. There is a set of ELROS statements used to define interfaces, which are text files input to the ELROS preprocessor. When the ELROS preprocessor compiles a program implementing invoker and/or performer interactions, the program must specify the interfaces it uses. We say a program *imports* the interfaces it needs to carry out its service interactions. There are several ELROS statements that are concerned with defining and using service interfaces.

Finally, interactions between an invoker and performer can generate various errors. For example, if the communications network that connects the machines of the invoker and performer fails, it may not be possible for an invoker to make a service request or for a performer to return the results of a request. As another example, if an invoker mistakenly uses an interface definition different from the one used by the performer, various errors are possible, such as service requests unsupported by the performer, results incomprehensible to the invoker and parameters of the wrong identity or type. ELROS supports an exception facility that allows the programmer to specify actions to take when such errors occur.

2.2 The Structure of ELROS programs

In addition to specifying a remote operations interface, an ELROS programmer embeds ELROS executable statements into the C language program of an invoker or performer. When the invoker or performer is compiled, the files containing the interface specification and the embedded language program are given as inputs to the ELROS preprocessor. The preprocessor produces a C program that is compiled by a C compiler to produce an executable (fig. 3).

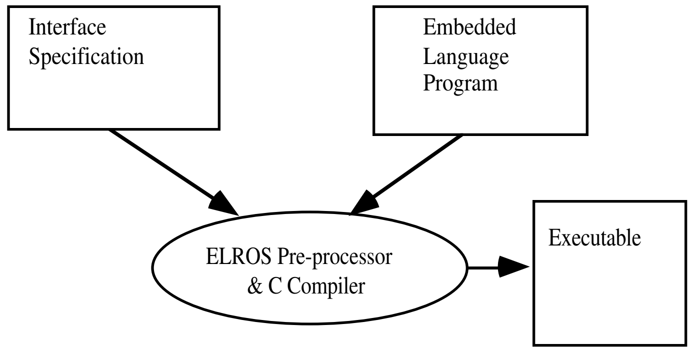

Figure 3 - ELROS preprocessor input and output

An ELROS interface has the following general form:

interface *interface-name optional-information*;

type-definitions

operation-definitions

end;

The reserved words **interface** and **end** are used to bracket interface definitions. Each interface is given a name (*interface-name*) that identifies it[4]. The *optional-information* is described in chapter 3. Interface definition statements are either type definitions or operation definitions, the latter corresponding to the specification of individual service requests[5].

2.3 Defining basic ASN.1 data types

Type definitions allow an interface designer to describe the data structures used by operations. Each type definition in the type definitions

---

[4] Each interface name imported into an invoker or performer must be unique. There are ways, using C preprocessor commands, to rename an interface within an embedded language program, thereby circumventing problems associated with duplicate interface names.

[5] At present ELROS interfaces cannot be nested (we intend to correct this problem in a future version). However, there is a way to achieve the same effect. If ELROS *type-definitions* and *operation-definitions* are placed in a file and the C preprocessor #include command is used within the body of an

section is terminated by a semicolon. There are 17 basic data types and four type constructors, the latter being used to create Arrays, Records and Unions. The basic data types are :

Boolean	- a value either **true** or **false**
Integer	- a signed integer value
Real	- a signed floating point value
BitString	- a value consisting of an array of bits
OctetString	- a value consisting of an array of octets
Enumerated	- a discrete set of values defined by the programmer
ObjectIdentifier	- a value consisting of an array of non-negative integers
Null	- no value
Any	- any value[6]
NumericString	- a character string value of numeric characters
PrintableString	- a character string value of printable characters
TeletexString	- a character string value of teletex characters
VideotexString	- a character string value of videotex characters
VisibleString	- a character string value of visible characters
IA5String	- a character string value of characters from the IA5 set
GraphicString	- a character string value of graphic characters
GeneralString	- a character string value of general characters

The last 8 data types are string types limited to characters from various standard alphabets. However, ELROS does not check to ensure that variables of these types contain only characters from the specified alphabets.

With two exceptions, the specification of a basic ELROS type requires not only identifying the base type (e.g., **Integer**, **BitString**), but also specifying the range of values supported by the type (for the moment we ignore an alternative form for "string types" described below). This is accomplished by

---

interface specification to include these definitions, the same *type-definitions* and *operation-definitions* can be used in multiple interfaces.

concatenating to the base type identifier a positive integer (or in some cases a pair of positive integers), contained within parentheses. In most cases the integer specifies the number of octets used to represent the type. For example, an integer type capable of containing values in the range $(-2^{31}, 2^{31}-1)$ would be defined as Integer(4). This follows because in twos-complement arithmetic[7], it takes 32 bits (i.e., 4 octets) to represent integers in that range. Similarly, an OctetString type capable of holding 16 octets would be defined as OctetString(16). The meaning of the positive integer(s) attached to the basic type names are :

Boolean(n)	**n** is the number of octets used to represent the boolean value (all bytes equal to zero representing **false**, all other values representing **true**).
Integer(n)	**n** is the number of octets used to represent the twos-complement integer.
Real(m, e)	**m** is the number of bits used to represent the mantissa and **e** is the number of bits used to represent the exponent. ELROS interprets these numbers as specifying a base 2 normalized format in which the exponent is biased half-way between its range.
BitString(b)	**b** is the number of bits in the BitString.
OctetString(n)	**n** is the number of octets in the OctetString.
Enumerated(n)	**n** is the number of octets used to represent the discrete values.
ObjectIdentifier(c, o)	**c** is the maximum number of components in the ObjectIdentifier and **o** is the number of octets per component.
Null	Null represents a type with no value, so no integer is specified.

---

[6] The Any type is special, since in practice it is the union of all types. Any values must be encoded and decoded by special macros that are discussed in chapter 5.

Any	Any represents an indeterminate type, so no integer is specified.
<chartype>String(n)	**n** is the number of characters in the character string.

The ELROS preprocessor translates each of these types into C typedef equivalents. This translation is controlled by the *parameters.dat* file read by the preprocessor each time it runs. For a typical machine the ELROS types might be translated according to the following rules (NB: ceiling(*expression*) represents the smallest integer greater than or equal to *expression*):

Boolean(n)	Assume a **short** is represented as two bytes, an **int** as four bytes, and a **long** as eight bytes. Boolean(1) -> Boolean(2) translates to **typedef unsigned short**, Boolean(3) -> Boolean(4) translates to **typedef unsigned int**, Boolean(5) -> Boolean(8) translates to **typedef unsigned long**, and Boolean(n), where n > 8, translates to **typedef struct { unsigned char val[n] ;}**.
Integer(n)	Assume a **short** is represented as two bytes, an **int** as four bytes, and a **long** as eight bytes. Then Integer(1) -> Integer(2) translates to **typedef short**, Integer(3) -> Integer(4) translates to **typedef int**, Integer(5) -> Integer(8) translates to **typedef long**, and Integer(n), where n > 8, translates to **typedef struct { unsigned char val[n]; }**.

[7] Signed integers in ELROS are assumed to be representated in twos-complement arithmetic.

Real(m, e)	Assume **float** represents a number with 24 mantissa bits and 8 exponent bits, and **double** represents a number with 53 mantissa bits and 11 exponent bits. Then Real(m,e), where m≤24 and e≤8, translates to **typedef float**; Real(m,e), where either 24<m≤53 and e≤11 or m≤53 and 8<e≤11, translates to **typedef double**; and Real(m,e), where either m > 53 or e > 11 translates to **typedef struct { unsigned char val[x]; }**, where x = ceiling(m/8) + ceiling(e/8) + 1.
BitString(b)	translates to **typedef struct { unsigned long len; unsigned char val[n] ;}**, where n = ceiling(b/8).
OctetString(n)	translates to **typedef struct { unsigned long len; unsigned char val[n] ;}**.
Enumerated(n)	Assume a **short** is represented as two bytes, an **int** as four bytes, and a **long** as eight bytes. Then Enumerated(1) -> Enumerated(2) translates to **typedef short**, Enumerated(3) -> Enumerated(4) translates to **typedef int**, Enumerated(5) -> Enumerated(8) translates to **typedef long**, and Enumerated(n), where n > 8, translates to **typedef unsigned char [n]**.
ObjectIdentifier(c, o)	translates to **typedef struct { int size, unsigned long comp[c];}**. **size** represents the number of components in the ObjectIdentifier. **o** must be less than or equal to sizeof(unsigned long).
Null	translates to **typedef void ***.
Any	translates to **typedef struct { unsigned char *pointer; unsigned long count; unsigned long size;}**. *pointer* holds the address of the buffer containing the Any value, *count* is the number of bytes in that buffer used to represent the value, *size* is the length of the buffer, and *sig* is used for internal checking.

<NULTerm>String(n) translates to **typedef char [n+1]**, where NULTerm is either Numeric, Printable, Visible, or Graphic. Strings of zero length are represented by a single NUL character.

<LenTerm>String(n) translates to **typedef struct { unsigned long len; char val[n];}** where LenTerm is either Teletex, Vidoetex, IA5, or General.

The distinction between NULTerm and LenTerm character string types is that NULTerm types are terminated by the NUL character, while LenTerm types, which may validly contain NUL characters, are terminated by the value in the “len” field.

There is an optional representation for certain ELROS basic types, called the *string types*, which consist of **BitString**, **OctetString**, **ObjectIdentifier** and the 8 character string types. This representation substitutes an indefinite value (represented by the character “*”) in place of the string length or number of ObjectIdentifier components. The ELROS preprocessor translates these indefinite length representations into the following C typedef equivalents.

BitString(*) translates to **typedef struct { unsigned long len; unsigned char *val; }**, where **len** = ceiling(b/8), b is the number of bits, and **val** holds the address of the BitString value.

OctetString(*) translates to **typedef struct { unsigned long len; unsigned char *val; }**, where **val** holds the address of the OctetString value.

ObjectIdentifier(*, o) translates to **typedef struct { int size, unsigned long *comp;}**. **size** represents the number of components in the ObjectIdentifier. **o** must be less than or equal to sizeof(unsigned long) and **comp** holds the address of the ObjectIdentifier value.

<NULTerm>String(*)	translates to **typedef char ***, where NULTerm is either Numeric, Printable, Visible, or Graphic. Strings of zero length are represented by a pointer to a single NUL character, not by a NULL pointer.
<LenTerm>String(*)	translates to **typedef struct { unsigned long len; char *val;}** where LenTerm is either Teletex, Vidoetex, IA5, or General.

To use variables of these types, the programmer must ensure that the relevant variable pointer references memory holding the variable value. We cover the use of the indefinite length types in more detail below.

2.4 Type constructors

The ELROS type constructors allow the programmer to define types in an interface specification that can be used as arguments in remote operation requests and replies within an embedded language program. While declarations using one of the basic ELROS types may appear in embedded language programs as well as interface specifications, ELROS type constructors may only appear within interface specifications.

2.4.1 Defining Arrays

Arrays of data elements are defined using the **Array** type constructor :

Array *element_type* *name_of_array_type* [*number_of_elements*]

Element_type may be any type, including one of the basic types as well as a type defined previously in the interface specification. The programmer identifies the Array type with *name_of_array_type*, which follows the *element_type*. The *number_of_elements* in the Array is given in square brackets following *element_type* and *name_of_array_type*. By default

ELROS represents an Array at the presentation layer as an ASN.1 SEQUENCE OF[8].

An ELROS declaration of this type translates to the following typedef:

typedef *element_type name_of_array_type*[*number_of_elements*]

For example, consider the following ELROS declaration:

```
Array Integer(4) vector[1000];
```

This defines a type with the name "vector". Variables of this type are capable of holding 1000 Integer(4) values. A declaration of a variable of this type in an embedded language program is made as follows:

```
vector sorted_list;
```

The ELROS preprocessor translates this into a C declaration that is equivalent to the following text (assume Integer(4) translates into the C type **int**):

```
int sorted_list[1000];
```

There is another array constructor available, which is useful when a programmer wishes to manage arrays on the heap, rather than on the stack. Arrays of this kind are defined using the **DynamicArray** type constructor :

DynamicArray *element_type name_of_array_type*

Element_type may be any type, including one of the basic types as well as a type defined previously in the interface specification. The programmer identifies the DynamicArray type with *name_of_array_type*, which follows the *element_type*. By default ELROS represents a DynamicArray at the presentation layer as an ASN.1 SEQUENCE OF.

---

[8] In chapter 3 we describe how to override this as well as other defaults.

This ELROS type constructor translates to the following typdef :

```
typedef struct name_of_array_type
        {
        unsigned long len;
        element_type *val;
        } name_of_array_type;
```

For example, suppose the following type declaration appears in an interface specification:

```
DynamicArray RelativeDistinguishedName RDNSequence;
```

This defines a type with the name RDNSequence. Suppose the following declaration appears in an embedded language program:

```
RDNSequence path;
```

The ELROS preprocessor translates it into the following C declaration:

```
struct
    {
    unsigned long len;
    RelativeDistinguishedName *val;
    } path;
```

A DynamicArray is treated differently by the ELROS generated code[9] depending on whether the value it contains is being sent, i.e., used in a **give** statement, or received, i.e., used in an **accept** statement (see section 2.8 for a description of **give**, and **accept** statements). When being sent, the ELROS generated code assumes the programmer has stored the appropriate values in “len” and “val”. That is, the ELROS generated code treats the memory (which may be either on the stack or on the heap) pointed to by “val” as a

[9] We use the phrase “ELROS generated code” to represent the combination of generated code and runtime system library code.

buffer containing “len” elements of *element_type*. In the example, the runtime system assumes “val” points to a buffer containing “len” RelativeDistinguishedName elements.

When the ELROS generated code is receiving into a DynamicArray, on the other hand, it allocates enough space from the heap (using malloc) to hold the elements sent and stores the address in “val”. Consequently, any pointer value in “val” is clobbered on a receive. The “len” field of the DynamicArray struct is set to the number of elements received.

Since the ELROS generated code allocates space from the heap for DynamicArrays, the programmer must free it before losing track of the values in “len” and “val”. To make this more convenient, the ELROS preprocessor generates free routines for DynamicArray types and for all types that contain one or more DynamicArray components (i.e., a Record or Union containing a DynamicArray component or a component that has a DynamicArray nested somewhere within it). Access to these free routines is possible by using the “rofreei()” and “rofree()” routines.

Rofreei() is called with the following parameters :

```
rofreei("type", &variable, interface)
```

The first argument is the ELROS type of the variable. Note that the type is placed within double quotes. The second argument is the address of the variable containing heap space to be freed. The third argument is the name of the interface from which the type is drawn (see section 2.7 for a description of interface definition and interface names). Note that the interface name is not placed within double quotes. A call to rofreei() frees the space allocated either by the programmer or by the runtime system associated with “variable”.

The ELROS runtime system must know the interface from which a type is drawn in order to call the correct routine to free its space (e.g., two interfaces imported into the embedded language code might define types of the same name). Rofreei() specifies the interface name through its third argument. The ELROS runtime system remembers this interface name in an internal variable, allowing a subsequent call of rofree() :

```
rofree("type", &variable)
```

The first argument is the ELROS type of the variable. Again, notice that the type is placed within double quotes. The second argument is the address of the variable containing heap space to be freed.

To illustrate how rofreei() and rofree() might be used, consider the variable “path” declared above. Assume it has been defined in an interface with the name “X_500_DAP_interface”. When the user sets the “len” and “val” fields of path, he normally would allocate space from the heap to hold the DynamicArray element values. After sending the values, he could free the space he allocated by calling :

rofreei(“RDNSequence”, &path, X_500_DAP_interface);

If he subsequently received a value into “path”, after using the values it references, he could free the space allocated from the heap by the ELROS runtime system by calling :

rofree(“RDNSequence”, &path);

Eliminating the interface name is possible because he previously called rofreei().

2.4.2 Defining Records

ELROS also provides a way to define Record types. The ELROS **Record** type constructor syntax is very much like the struct syntax in C :

Record { *Record-list* } *Record-type-name*;

All Record type definitions begin with the reserved word **Record**, followed by a semicolon-separated list of Record element definitions (*Record-list*) contained within braces and then by the name of the Record type (*Record-type-name*), terminated by a semicolon. The simplest element definition is of the following form :

type-name *element-name*;

Record elements also may be definitions of other Records, Arrays or Unions. By default ELROS represents a Record at the presentation layer as an ASN.1 SET.

For example, consider the following ELROS Record type declaration:

```
Record
    {
    ObjectIdentifier(20,4) type;
    DynamicArray Any values;
    } Attribute;
```

This defines a type with the name "Attribute" consisting of an ObjectIdentifier and a DynamicArray of Any values. The ELROS preprocessor would translate this into the following C typedef:

```
typedef
    struct
        {
        struct /* ObjectIdentifier */
            {
            int size;
            unsigned long comp[20]
            } type;
        struct /* DynamicArray */
            {
            unsigned long len;
            struct any /* Any */
                {
                unsigned char *pointer;
                unsigned long count;
                unsigned long size;
                struct any * sig;
                } val;
            } values;
        } Attribute;
```

2.4.3 Defining Unions

ELROS allows the definition of data types that are Unions of other types. A Union type is defined with the **Union** type constructor :

Union {*Union-list*} *Union-type-name*;

All Union type definitions begin with the reserved word **Union**, followed by a semicolon-separated list of Union elements (*Union-list*) contained within braces and then by the name of the Union type(*Union-type-name*), terminated by a semicolon. The simplest element definition is of the following form :

type-name   *element-name*;

Union elements also may be definitions of other Records, Arrays or Unions. ELROS represents a Union at the presentation layer as an ASN.1 CHOICE.

Union types in C are non-discriminated. That is, there is no information available at runtime that specifies which element is held within the Union. ELROS Union types, on the other hand, are discriminated, since there must be a way to identify the valid Union element in order for the ELROS runtime system to encode or decode it properly. Consequently, ELROS Unions map to a C struct with two elements – one the Union specified in the ELROS Union definition and the other a tag that specifies which element is present in the Union.

For example consider the following ELROS Union type declaration:

```
Union
    {
    Attribute               addAttribute;
    ObjectIdentifier(10,4)  removeAttribute;
    Attribute               addValues;
    Attribute               removeValues;
    } EntryModification;
```

ELROS maps this declaration into the following C typedef :

```
typedef
    struct
            {
            int tag;
            Union
                  {
                  Attribute                   addAttribute;
                  ObjectIdentifier(10,4)      remoteAttribute;
                  Attribute                   addValues;
                  Attribute                   remoteValues;
                  } un;
            } EntryModification;
```

Here the ELROS types "Attribute" and "ObjectIdentifier(10, 4)" are used in the EntryModification definition, rather than showing their mapping into a C struct, in order to maintain pedagogical clarity.

To send a variable of this type to a correspondent, EntryModification.tag must be set to the number of the Union element present in the C non-discriminated union. Elements are numbered consecutively beginning at 0. So, the Union element addAttribute is element 0, removeAttribute is element 1, etc. When a Union variable receives information from a correspondent, the tag is set to the number of the Union element received.

2.5 Operation definitions

There are four operation services in the remote operations standard : 1) invokes, 2) results, 3) errors and 4) rejects. ELROS allows the definition of operations from all of these service classes. We only cover invokes, results and errors here. The definition and use of rejects is described in chapter 4.

The general format for an operation definition in an ELROS interface is:

service_class operation_name operation_number
parameter

The valid values for *service_class* are : 1) **invoke**, 2) **result**, and 3) **error**. The *operation_name* may be any valid C identifier and the *operation_number* may be either an integer or ObjectIdentifier constant. The *parameter* is either an ELROS data type defined previously in the interface or one of the basic ELROS data types[10]. The *parameter* field of an operation definition is optional.

For example, to define the operations necessary to read an X.500 directory entry, we might use the following interface definitions:

```
invoke ReadOp                    1
        ReadArgument;

result ReadResultOp              1
        ReadResult;

error AttributeErrorOp           1
        AttributeErrorParameter;

error NameErrorOp                2
        NameErrorParameter;

error ServiceErrorOp             3
        ServiceErrorParameter;

error ReferralOp                 4
        ReferralErrorParameter;

error AbandonedOp                5;

error SecurityErrorOp            6
        SecurityErrorParameter;
```

[10] ELROS supports operation definitions with multiple parameters. However, these parameters are enclosed within ASN.1 SETs and so are not compatible with most ISO standard application protocol definitions.

These definitions assume the data types ReadArgument, ReadResult, AttributeErrorParameter, NameErrorParameter, ServiceErrorParameter, ReferralErrorParameter, AbandonedErrorParameter, and SecurityErrorParameter are previously defined in the interface specification.

2.6 Translating ASN.1 into ELROS data structures

We are now in a position to illustrate how an ELROS interface specification for an ISO Application protocol might appear. We will use an ASN.1 fragment from the X.500 recommendations for this purpose. The ASN.1 definitions for this fragment are found in Appendix A. We draw certain definitions from the fragment and show their corresponding ELROS data structures.

We have not yet explained how a programmer changes the default rules used to encode and decode an ELROS data structure into ASN.1. For example, an X.500 Attribute type is encoded as an ASN.1 SEQUENCE. However, an Attribute is represented in ELROS as a Record, which by default is encoded as an ASN.1 SET. Similarly, each element of the Attribute type has a unique context specific tag, which is not shown in the ELROS Record. In chapter 3 we discuss how to annotate the data structure in order to control its encoding rules. For the moment we ignore this problem and show the ELROS data structures as if the default rules produced the correct encodings.

The ReadArgumentType definition shown in Appendix A reads :

```
ReadArgumentType        ::=     SET {
                                        object [0] Name,
                                        selection [1] EntryInformationSelection
                                                DEFAULT {},
                                        [30] ServiceControls DEFAULT {},
                                        [29] SecurityParameters DEFAULT {},
                                        requestor [28]
                                                DistinguishedName OPTIONAL,
                                        [27] OperationProgress
                                                DEFAULT {notStarted}
```

```
                                        aliasedRDNs [26]
                                            INTEGER OPTIONAL,
                                        extensions [25] SET OF Extension
                                            OPTIONAL }
```

The ELROS data structure corresponding to this definition is :

```
Record
    {
    Name object;
    EntryInformationSelection selection;
    ServiceControls serviceControls;
    SecurityParameters securityParameters;
    DistinguishedName requestor;
    OperationProgress operationProgress;
    Integer(4) aliasedRDNs;
    DynamicArray Extension extensions;
    } ReadArgumentType;
```

Comparing the two definitions, there are several things to note. First, the order in which types and element names appear is switched. ASN.1 puts the name of the SET element first and its type second. ELROS puts the type first and the element name second. We chose this ordering for ELROS because it is the same as that used for struct definitions in C, the host language in which ELROS is embedded.

Secondly, none of the ASN.1 information specifying context specific tags, DEFAULTs and OPTIONAL elements is present in the ELROS definition given above. The annotations required to specify this information are covered in chapter 3.

Thirdly, some of the ASN.1 SET elements are specified only by their types without an associated element name. Elements in an ELROS Record definition must be given an associated name.

Finally, when specifying the aliasedRDNs element in the ELROS definition, a length for the Integer is given. The corresponding ASN.1 simply specifies this element is an INTEGER, which can be of arbitrary size. The programmer must decide a reasonable bound for the range of any basic type used in an interface definition.

We will examine two elements of ReadArgumentType in order to illustrate other factors in building ELROS data structures from ASN.1 specifications. The ASN.1 definition of EntryInformationSelection is :

```
EntryInformationSelection       ::=     SET {
                                            attributeTypes CHOICE {
                                              allAttributes [0] NULL,
                                              select [1] SET OF AttributeType }
                                                    DEFAULT allAttributes NULL,
                                            InfoTypes [2] INTEGER
                                                    {attributeTypesOnly (0),
                                                     attributeTypesAndValues (1)}
                                                     DEFAULT
                                                           attributeTypesAndValues }
```

The corresponding ELROS type definition is :

```
Record
    {
    Union
           {
            NULL allAttributes;
            DynamicArray AttributeType select;
            } attributeTypes;
    Integer(4) InfoTypes;
    } EntryInformationSelection;
```

This example is notable because it uses an embedded Union definition within the Record definition corresponding to the CHOICE in the ASN.1 specification.

The second element of ReadArgumentType that we will examine is OperationProgress. Its ASN.1 specification is :

```
OperationProgress               ::=     SET {
                                            nameResolutionPhase [0]
                                                    ENUMERATED {
                                                           notStarted (1),
```

```
                                        proceeding (2),
                                        completed (3) },
                                nextRDNToBeResolved [1]
                                        INTEGER OPTIONAL }
```

The corresponding ELROS type definition is :

```
Record
    {
    Enumerated(1)  nameResolutionPhase;
    Integer(1) nextRDNToBeResolved;
    } OperationProgress;
```

ReadArgumentType is used in the definition of ReadArgument as follows :

```
ReadArgument            ::=     CHOICE {
                                        ReadArgumentType,
                                        SEQUENCE {
                                                ReadArgumentType,
                                                AlgorithmIdentifier,
                                                BIT STRING }  }
```

This translates to the following ELROS data structure :

```
Union
    {
    ReadArgumentType  read_argument;
    Record
            {
            ReadArgumentType  readArgument;
            AlgorithmIdentifier  algorithmIdentifier;
            BitString(2048) signature;
            } signatureBlock;
    } ReadArgument;
```

These data structures are used in chapter 3 to demonstrate how to specify an ELROS interface used in an implementation of an ISO Application protocol.

2.7 Specifying an ELROS interface

As described in section 2.2, an ELROS interface is a set of data structure and operation definitions collected into a named unit. The format of an interface is :

interface *interface-name optional-information*;
  type-definitions
  operation-definitions
end;

For example, an interface using the data structures we have developed so far might look like :

```
interface  X_500_DAP_interface;

    Record
        {
        Union
                {
                 NULL allAttributes;
                 DynamicArray AttributeType select;
                 } attributeTypes;
        Integer(4) InfoTypes;
        } EntryInformationSelection;

    Record
        {
        Ennumerated(1) nameResolutionPhase;
        Integer(1) nextRDNToBeResolved;
        } OperationProgress;
```

```
Record
    {
    Name object;
    EntryInformationSelection selection;
    ServiceControls serviceControls;
    SecurityParameters securityParameters;
    DistinguishedName requestor;
    OperationProgress operationProgress;
    Integer(4) aliasedRDNs;
    DynamicArray Extension extensions;
    } ReadArgumentType;

Union
    {
    ReadArgumentType readArgument;
    Record
            {
            ReadArgumentType readArgument;
            AlgorithmIdentifier algorithmIdentifier;
            BitString(2048) signature;
            } signatureBlock;
    } ReadArgument;

invoke ReadOp 1
        ReadArgument;

result ReadResultOp 1
        ReadResult;

error AttributeErrorOp 1
        AttributeErrorParameter;

error NameErrorOp 2
        NameErrorParameter;

error ServiceErrorOp 3
```

```
            ServiceErrorParameter;

    error ReferralOp 4
            ReferralErrorParameter;

    error AbandonedOp 5;

    error SecurityErrorOp 6
            SecurityErrorParameter;

end
```

Obviously, the interface definition given above is missing a significant portion of its definitions. Most of the data structure definitions for ReadOp are missing as well as all of the data structures for the remaining operations. Appendix B contains the complete ELROS interface specification for the X.500 fragment found in Appendix A.

2.8 Using an ELROS interface

Once we have defined an interface, we are in a position to write an invoker and performer that uses its operations. There are two activities that must be programmed for invoker-performer interactions: 1) making contact between them, and 2) making and handling service requests.

Since distributed application components may be engaged in several interactions concurrently, there must be some way to discriminate between them. In ELROS we use the notion of a communications instance to do this. Communications instances are much like open files, and communications instance variables are similar to file pointers in that they represent the state of an on-going interaction. In particular, a communications instance represents an active connection between the invoker and performer. Programs are allowed to have many communications instances active concurrently and are even allowed to have multiple communications instances to the same invoker or performer concurrently active.

A communications instance variable is declared with the **control** reserved word. For example the following ELROS statement declares two instance variables (to economize writing style, we often refer to communications instance variables simply as instance variables), inst1 and inst2 :

```
control(X_500_DAP_interface)  inst1, inst2;
```

Notice in the instance variable declaration we have placed the word "X_500_DAP_interface" in parentheses after the reserved word **control**. This is the name of the interface that includes the definitions of operations that will be used over the instance.

Since the ELROS preprocessor must be able to read and process this interface, we must tell it the name of the file in which the interface is located. This is called **importing** the interface and we do this with an include statement[11]. Suppose the interface "X_500_DAP_interface" is stored in the file "~/dist/interfaces/X_500_DAP_fragment.i". This interface is made known to the ELROS preprocessor by including in the embedded language program the statement :

```
#include  "~/dist/interfaces/X_500_DAP_fragment.i"
```

The include statement must appear before the declaration of any instances that use the X_500_DAP_ interface.

The include statement and instance declaration are the first statements we have described that are inserted by the programmer into C programs directly, rather than appearing in interface specifications. Interface specifications usually are located and maintained in files separate from those containing the ELROS embedded language programs in order to share them between different programs.

2.8.1 Writing the invoker

[11] It is possible to place an interface specification in the same file as, and textually preceding, an embedded language program. However, this is not recommended, since doing so would require the programmer to maintain at least two copies of the interface specification, one in the invoker program and the other in the performer program.

Once we have imported the appropriate interface and declared an instance variable associated with it, we are in a position to program the interactions between an invoker and performer. In ELROS a component uses the **bind_addr** statement to contact a correspondent. **bind_addr** takes four arguments : 1) an instance variable, 2) rules, 3) a pointer to an address structure, and 4) a timeout value. The instance variable is the output parameter of **bind_addr**, being set to a value that can be used by other ELROS statements (much like a file pointer is initialized by an fopen call). The rules argument is an **enum** type with only two choices at present, asn_1_ber and asn_1_ler. The timeout value is used to timeout both ELROS statements that block (including **bind_addr** statements) as well as the underlying communication routines in the ELROS runtime system.

The address structure argument contains information that identifies the performer. It has the following general format:

```
struct address
    {
    bind_platform platform; /* isode, puresockets */
    bind_role role; /* initiator, responder */
    union
          {
          struct
                {...

                } isode_addr;

          struct
                {...

                } puresocket_addr;
          } un;
    };
```

This structure is available as a typedef with the identifier **bind_address**. (**NB**: not all elements of the address structure are shown above.

Consequently, application programs should always use the **bind_address** typedef when declaring address structures).

The address structure argument must be filled in by the invoker or performer before calling **bind_addr**. To ease the programming chore this imposes, platform dependent routines are provided that initialize the address structure and fill in the appropriate content. For the ISODE platform this routine is **init_isode_addr**(&addr, role, <address_string>). The first argument is a pointer to the address structure. The second argument specifies the role the component is taking in interactions over the communications instance[12]. In the case of an invoker, this role is initiator[13]. The third argument is a string in one of two formats. When the programmer wishes to specify only the IP address and port (i.e., the programmer is willing to use the default transport, session and presentation selectors[14]), the string consists of the substring “Internet=“ followed by the dotted IP address, a "+" character and a port value. **Init_isode_addr** fills in the IP address and port into the appropriate position in the ISODE address structure and sets the transport, session and presentation selectors to the default values.

The second format allows the programmer to specify presentation, session and transport selectors. Each selector is represented by a “#” character followed by a decimal value, which must be in the range 0 to 65,535. The presentation selector is given first, followed by the session selector and then the transport selector. The presentation and session selectors are optional. Following the selector field(s) are the IP address and port values, which are separated by a “+” character and preceded by the string “Internet=“. The selector fields are delimited by a “/” character, which also separates the transport selector field from the IP address and port field. For example, the string “#0/#0/#521/Internet=

---

12 The “role” argument is of type “bind_role”, which is an enum with constants “initiator” and “responder”.

13 Actually, this is not necessarily true. ELROS makes no restriction on which correspondent may be an initiator and which a responder. However, in most applications the invoker is the inititator and the performer is the responder. Both “initiator” and “responder” are values of an enumeration.

14 The default transport selector is 1024, the default session selector is 0, and the default presentation selector is 0.

134.8.20.12+17003” represents an ISODE address with zero presentation and session selectors, a transport selector of 521, an IP address of 134.8.20.12 and a port of 17003.

To illustrate how init_isode_addr() would be used in a program, consider an invoker that contacts a performer implementing the ReadOp operation shown above. If the invoker and performer interact using the ISODE communications platform and the performer is at IP address 134.8.20.12 and port 17003, the invoker code to contact the performer would look something like the following (assuming a timeout value of 10 seconds) :

```
#include "~/dist/interfaces/X_500_DAP_fragment.i"

control(X_500_DAP_interface) X_500_performer;
bind_address addr;

init_isode_addr(&addr, initiator, "Internet=134.8.20.12+17003");

bind_addr(X_500_performer, asn_1_ber, &addr, 10);
```

Once a communications instance is bound by executing **bind_addr**, it can be used by the invoker to request a ReadOp operation with the ELROS **give** statement, which has the following form :

give *instance-variable C-statement*

The **give** statement begins with the reserved word **give**, followed by an instance variable and then by a C language statement (which may be a compound statement and include other ELROS embedded language declarations and executable statements). To send an operation to the performer, the programmer embeds a reference to it in the C statement, giving within parentheses an expression for its parameter. For example, sending the ReadOp operation to an X.500 performer could be accomplished with the following embedded ELROS code :

```
ReadArgument read_arg;
```

```
/* fill in read_arg */
          .
          .
give  X_500_performer
      {
      ReadOp(read_arg);
      }
```

In this simple example, only one operation appears within the **give** statement. However, it is possible to **batch** remote operations in a single presentation protocol data unit (PPDU) by including multiple operation references in a single **give** statement. Each reference to an operation that occurs within the context of a **give** statement is placed in a separate presentation data value (PDV) of the PPDU. The PPDU is sent when control passes from the **give** statement.

Operation references that occur within a loop or another control structure embedded within the **give** statement cause the creation of a new PDV each time the operation reference is executed. This provides the programmer with a tool to optimize communications performance when a series of operations are sent to the performer before a result is required.

To receive the results of an operation, an invoker uses the **accept** statement, which has the following form :

accept *instance-variable*

{

on $operation_1(parameter_1)$

$C\text{-}statement_1$

on $operation_2(parameter_2)$

$C\text{-}statement_2$

.

.

.

on *operation(parameter)*

$C\text{-}statement_n$

}

The **accept** statement is tied to a particular communications instance by the instance variable given immediately following the **accept** reserved word. Following the instance variable and contained within opening and closing curly braces is a series of on-clauses, each of which starts with the reserved word **on**, that specify an operation (in this case a **result** operation) in the interface to which the instance variable is tied by its declaration. The operation specified in each on-clause takes a *parameter*, which specifies the variable into which the ELROS runtime system stores the value communicated by the correspondent. Note there is no punctuation mark following the line containing the reserved word **on**, the operation and its associated parameter. In this way the syntax of an **accept** statement differs from that of a **switch** statement, which terminates the **case** statement with a colon. The *C-statement* following each operation specification can be any general C language statement, including a compound statement. It may contain not only C programming constructs, but also other ELROS embedded language statements, including declarations, **bind_addr** statements, **accept** statements, and **give** statements.

When the **accept** statement is executed, the program blocks waiting for an operation to arrive over the interface. When an operation is received, control passes to the appropriate on-clause. Control passes out of the **accept** statement if an operation doesn't arrive within the timeout period specified in the **bind_addr** call. This error condition can be detected using the **exceptions** clause described in chapter 4.

After sending the ReadOp operation to the performer and obtaining the results, the invoker should let the performer know that no more operations will arrive over the communications instance (this is like closing an open file). Consequently, the invoker should unbind the instance with the ELROS **unbind** primitive before proceeding. Unbind takes one argument, a bound instance variable, e.g.,.

```
unbind(X_500_performer);
```

The complete program that contacts an X.500 performer, reads an X.500 entry and unbinds the communications instance is :

```
/* import the X_500 interface */
```

```
#include "~/dist/interfaces/X_500_DAP_fragment.i"

/* declare the instance, address structure variables and operation
    parameters */

    control(X_500_DAP_interface) X_500_performer;
    bind_address addr;
    ReadArgument read_argument;
    ReadResult read_result;

/* initialize the address structure and fill in the performer's address */

    init_isode_addr(&addr, initiator, "Internet=134.8.20.12+17003");

/* establish communications with the performer */

    bind_addr(X_500_performer, asn_1_ber, &addr, 10);

/* fill in read_arg */
                .
                .
                .

/* make a read request of the performer */

    give X_500_performer
          {
          ReadOp(read_argument);
          }

/* wait for the read_result */

    accept X_500_performer
          {
          on ReadResultOp(read_result)
```

```
        /* process the read_result */
                .
                .
                .
    }

/* unbind the communications instance */

    unbind(X_500_performer);
```

2.8.2 Writing the performer

A component acting in the role of a performer waits to be contacted by an invoker and then carries out the operations the invoker requests. Performers wait for contact by executing **bind_addr**, specifying the role of **responder**. As with an invoker, the performer must first initialize an address structure. For example, the following code would be typical for the X_500 performer used by the invoker described above :

```
/* import the X_500 interface */

#include  "~/dist/interfaces/X_500_DAP_fragment.i"

/* declare the instance, address structure variables and operation
    parameters */

    control(X_500_DAP_interface)  X_500_performer;
    bind_address  addr;
    ReadArgument  read_argument;
    ReadResult  read_result;

/* initialize the address structure and fill in the performer's address */

    init_isode_addr(&addr, responder, "Internet=134.8.20.12+17003");
```

```
/* establish communications with the invoker */

    bind_addr(X_500_invoker, asn_1_ber, &addr, 10);
```

Note that this program fragment is very similar to that executed by the invoker, except “init_isode_addr” is called with its role parameter set to "responder". Since the address structure role specifies responder, **bind_addr** will block waiting for contact from an initiator specifying this address.

Once contact is made with an invoker, the performer accepts service requests, carries out the activity to satisfy them and returns any results of the requests to the invoker. Normally, a performer supports more than one operation; although, in our simple example this is not so. Consequently, the performer must first determine which of its operations the invoker is invoking. The performer uses an accept statement to accomplish this.

When the **accept** statement is executed, the program blocks waiting for an operation to arrive over the instance. When an operation is received, control passes to the appropriate on-clause.

For example, the code to implement an X_500 performer might look like :

```
/* import the X_500 interface */

#include  "~/dist/interfaces/X_500_DAP_fragment.i"

/* declare the instance, address structure variables and operation
    parameters */

    control(X_500_DAP_interface)  X_500_invoker;
    bind_address  addr;
    ReadArgument  read_argument;
    ReadResult  read_result;

/* initialize the address structure and fill in the performer's address */

    init_isode_addr(&addr, responder, "Internet=134.8.20.12+17003");
```

```
/* establish communications with the invoker */

    bind_addr(X_500_invoker, asn_1_ber, &addr, 10);

/* obtain a ReadOp request and service it */

    accept X_500_invoker
        {
        on ReadOp(read_argument)
              {
              /* call the subroutine that performs a read on the X_500
                  database */

              read_database(&read_result, &read_argument);

              /* return the read result to the invoker */

              give X_500_invoker
                    {
                    ReadResultOp(read_result);
                    }
              }
        }

/* unbind the communications instance */

    unbind(X_500_invoker);
```

There are two notable things about this program fragment. First there is a distinct symmetry between the invoker and performer code. The **init_isode_addr** call is the same, except the invoker specifies a role of initiator, while the performer specifies a role of responder. The invoker contains **invoke** operations in a **give** statement that the performer accepts in on-clauses of an **accept** statement. Similarly, the performer contains **result** operations in a **give** statement that the invoker accepts in on-clauses of an **accept** statement.

Secondly, this program will accept exactly one operation and then terminate. In order to write a performer that potentially accepts more than one operation over a bound communications instance, the **accept** statement must be placed within a loop.

2.8.3 On Done

The invoker and performer developed in the previous sections have a significant deficiency. The invoker contacts the performer, reads an X.500 database entry, unbinds the communications instance and continues computing. The performer waits for a ReadOp request to arrive, accesses the indicated entry, returns the results read and then terminates. This strategy is not going to work well if the invoker wishes to read more than one entry. The strategy is also deficient if the performer interacts with more than one invoker.

The obvious solution to this problem is to program the invoker so it executes calls to the performer in a loop. The performer can also be programmed to wait for contact from an invoker and then loop servicing new ReadOp requests. The invoker program fragment that does this looks like :

```
/* import the X_500 interface */

#include  "~/dist/interfaces/X_500_DAP_fragment.i"

/* declare the instance, address structure variables and operation
    parameters */

    control(X_500_DAP_interface)  X_500_performer;
    bind_address  addr;
    ReadArgument  read_argument;
    ReadResult  read_result;

/* initialize the address structure and fill in the performer's address */

    init_isode_addr(&addr, initiator, "Internet=134.8.20.12+17003");
```

```
/* establish communications with the performer */

    bind_addr(X_500_performer, asn_1_ber, &addr, 10);

/* make 8 read requests of the performer */

    for(i=0; i<8; i++)
        {
        /* fill in read_arg */
                .
                .
                .

        /* make the next read request of the performer */

        give X_500_performer
              {
              ReadOp(read_argument);
              }

        /* wait for the read_result */

        accept X_500_performer
              {
              on ReadResultOp(read_result)

                /* process the read_result */
                        .
                        .
                        .
              }
        }

/* unbind the communications instance */
```

```
unbind(X_500_performer);

<remainder of program>
```

Writing the performer program fragment is a little more problematical, since, as yet, we have not described a way for the performer to discover when the invoker is finished with the communications instance (i.e., when the invoker executes an **unbind**). ELROS provides a mechanism for the performer to discover this, which is also used by both invokers and performers to react to errors. We concentrate here on how a performer is informed when an invoker unbinds an instance.

ELROS allows the attachment of an exceptions clause to many of its executable statements, including the **accept** statement. The exceptions clause begins with the reserved word **exceptions** and is followed by a set of on-clauses (much like the structure of an **accept** statement) that specify certain exceptional termination conditions, each followed by a (possibly compound) statement to execute when those conditions occur. The "done" condition represents an **unbind** by a correspondent (this condition is specified by the reserved word **on** followed by the reserved word **done**).

When the performer executes an **accept** statement with an attached **exceptions** clause containing a **done** on-clause and the invoker executes an **unbind**, control in the performer passes to the code immediately after the **on done** reserved words. For example, suppose we attached the following **exceptions** clause to the **accept** statement in the X_500 performer :

```
accept X_500_invoker
    {
    on ReadOp(read_argument)
          {
          /* call the subroutine that performs a read on the X_500
              database */

          read_database(&read_result, &read_argument);

          /* return the read result to the invoker */
```

```
            give X_500_invoker
                  {
                  ReadResultOp(read_result);
                  }
            }
      }
   exceptions
      {
      on done
            unbind(X_500_invoker);
      }
```

When the invoker unbinds the communications instance, control in the performer passes from the **accept** statement to the **done** on-clause within the **exceptions** clause, causing the performer to unbind its instance variable.

This still doesn't allow the performer to handle more than one ReadOp request in a single execution. The **accept** statement is not a looping statement. It executes once, either branching to the ReadOp on-clause, to the **done** on-clause or timing out. To accept more than one ReadOp request, the **accept** statement must be placed in either a **for** or a **while** loop. Using a **while** loop, the complete performer program fragment is :

```
#define FALSE 0
#define TRUE 1

   short bound;

/* import the X_500 interface */

#include  "~/dist/interfaces/X_500_DAP_fragment.i"

/* declare the instance, address structure variables and operation
   parameters */

   control(X_500_DAP_interface)  X_500_invoker;
```

```
    bind_address  addr;
    ReadArgument  read_argument;
    ReadResult  read_result;

/* initialize the address structure and fill in the performer's address */

    init_isode_addr(&addr, responder, "Internet=134.8.20.12+17003");

/* establish communications with the invoker */

    bind_addr(X_500_invoker, asn_1_ber, &addr, 10);

/* obtain the ReadOp requests and service them. When the invoker unbinds,
    unbind the communications instance and exit the while loop */

    bound = TRUE;

    while(bound)
        {
        accept X_500_invoker
            {
            on  ReadOp(read_argument)
                {
                /* call the subroutine that performs a read on the X_500
                    database */

                read_database(&read_result,  &read_argument);

                /* return the read result to the invoker */

                give  X_500_invoker
                    {
                    ReadResultOp(read_result);
                    }
                }
            }
```

```
exceptions
    {
    on done
        {
        unbind(X_500_invoker);
        bound = FALSE;
        }
    }
}
```

The details of the **while** loop may bother some programmers concerned with good programming style. They might suggest using a **break** statement within the **done** on-clause to exit the **while** loop. This would be a grave error. The ELROS preprocessor translates **accept** statements and **exceptions** clauses into **switch** statements. Consequently, the use of **break** within these statements is dangerous. **Break** may only be used to re-route control within a C language statement embedded in an ELROS statement (e.g., within a **while** loop embedded in the C language statement part of an on-clause). **Break** must never be used to re-route control outside of an **accept** statement or on-clause.

2.9 Advanced data structure techniques

ELROS supports a number of advanced features for programmers who require sophisticated data structures. In particular, ELROS supports recursive data structures, the use of nested type definitions (i.e., anonymous types), forward referencing of certain kinds of type definitions, the use of indefinite length variables, and heap space management routines to support recursive data structures, DynamicArrays, and indefinite length variables.

2.9.1 Recursion in Records and Unions

Most ISO Application protocol specifications utilize the capability of ASN.1 to define recursive data structures. Since one of the design goals of ELROS is to support ISO protocol implementations, the interface definition language allows recursive references within Record and Union data structures and as the element type of a DynamicArray. A recursive reference may appear nowhere else. Furthermore, the type being referenced must be either a Record or Union. Recursive references may be nested arbitrarily deep.

To define a recursive data structure, its identifier is used within its definition. For example, when specifying the SearchArgument for an X.500 Search operation, the programmer provides a filter data structure that controls the search activity. The filter data structure is defined in ASN.1 as:

```
Filter  ::=  CHOICE {
                 item   [0]  FilterItem,
                 and    [1]  SET OF Filter,
                 or     [2]  SET OF Filter,
                 not    [3]  Filter }
```

Note the reference to Filter in the "and", "or" and "not" components of Filter. Also note that Filter is referenced as a CHOICE item (i.e., as the type of "not") and as the element type of a SET OF constructor (i.e., within the type definitions for the "and" and the "or" components).

A recursive reference in ELROS is specified in one of two ways. If the identifier is used as the type of a Record or Union component, the recursive reference is indicated by appending the characters "(*)" to the type identifier. If the identifier is used as the element type of a DynamicArray that defines a Record or Union component type, these characters are not appended.

The difference in syntax exists because a recursive reference occurring as the type of a Record or Union component generates a struct pointer in the corresponding C data structure. However, a recursive reference occurring as the element type of a DynamicArray generates a struct pointer as the type of the "val" field within the struct corresponding to the DynamicArray. Consequently, recursive references within DynamicArray definitions do not cause any change in the architecture of the data structure (i.e., the "val" field is a pointer whether or not the DynamicArray element type is recursive.

The ELROS data structure that corresponds to the ASN.1 definition of Filter (assuming FilterItem is already defined) is :

```
Union
      {
       FilterItem <context 0> item;
      DynamicArray <context 1 set> Filter and;
      DynamicArray <context 2 set> Filter or;
      Filter <context 3> (*)not;
      } Filter;
```

Note that the recursive references to Filter occurring as the element type of the “and” and “or” DynamicArrays do not have the characters “(*)” suffixed, while the recursive reference that specifies the type of the “not” component does suffix these characters.

The ELROS preprocessor translates recursive references within an ELROS data structure to a pointer to the recursive type within the corresponding C struct or union. For example, the C data structure corresponding to the Filter definition (simplified somewhat for pedagogical reasons) is :

```
struct Filter
      {
      int tag;
      union
            {
            FilterItem item;
            struct
                  {
                  unsigned long len;
                  struct Filter *val;
                  } and;
            struct
                  {
                  unsigned long len;
                  struct Filter *val;
```

```
            } or;
        struct Filter * not;
        } un;
    } Filter;
```

The use of variables of a recursive type follows the model for DynamicArrays. That is, when they are received, the ELROS generated code allocates space to accept all of the data transmitted and places references to that space at the appropriate components within the variable. For example, when a variable of type Filter is received, the ELROS generated code allocates space to accept the recursive value and then places the appropriate references in the components “and”, “or” or “not”.

2.9.2 Anonymous types and DynamicArrays

It is sometimes desirable to include the definition of a type within the body of another type definition. For example, sometimes a Record is needed within a type, the Record is small (say, two or three components) and is used nowhere else. In this case it is convenient to simply embed the Record definition within the encapsulating type definition.

When a type is defined within another type, it cannot be referenced by name, since there is no provision in the ELROS data structure definition language for naming the internal type. Consequently, such types are said to be *anonymous*.

Normally, there is no problem with using anonymous types. References to their components are made through references to components of the encapsulating type. However, one situation does occur in which anonymous types pose a problem, i.e., when the element type of a DynamicArray is anonymous.

The problem occurs when a programmer needs to allocate space for the DynamicArray using malloc(). Since the element type is anonymous, there is no type name that the programmer can use to cast the pointer returned by malloc(), nor is there a type name to provide as the argument to sizeof() to compute the size of the DynamicArray in bytes.

To illustrate this problem consider the following two type definitions :

```
Record
    {
    Integer(4) i;
    Boolean(1) b;
    } InternalType;

DynamicArray InternalType array1;

DynamicArray Record { Integer(4) i; Boolean(1) b;} array2;
```

When allocating space for a variable of type array1, the programmer can use the identifier “InternalType” both as the cast type as well as the argument to sizeof(). However, such an identifier is unavailable for allocating space for array2. To be specific, the following code allocates 10 elements of the variable “a1” :

```
array1 a1;
a1.len = 10;
a1.val = (InternalType *) malloc(sizeof(InternalType) * 10);
```

However, no corresponding code can be written for a variable of type array2. This suggests that programmers should normally avoid the use of *anonymous* types in **DynamicArray** definitions.

2.9.3 Indirect type references

Consider the implementation of an n-ary tree, where “n” is unbounded. Each node in the tree contains references to “i” direct descendants as well as a datum associated with the node. Assume the datum is a character string.

One way to implement this data structure is to use a forward type reference. For example, the following ELROS defintions implement an n-ary tree :

```
DynamicArray Node Nodes;
Record
    {
    PrintableString(*) data;
    Nodes subtree;
    } Node;
```

Notice that a forward reference uses exactly the same syntax as a recursive reference (which is just a special case of a forward reference). Futhermore, the forward reference generates a **struct** pointer as the type of the “val” field, as does a recursive reference. For example, the C **typedef** produced by the ELROS preprocessor from the “Node” and “Nodes” type definitions are :

```
typedef struct Nodes
        {
        unsigned long len;
        struct Node *val;
        } Nodes;

typedef struct Node
        {
        char * data;
        Nodes subtree;
        } Node;
```

ELROS allows forward type references only in certain cases. The type being forward referenced must be either a **Record** or a **Union**. Also, the forward reference may occur only as a component of a **Record** or **Union** or as the element type of a **DynamicArray**.

Forward type references occurring as **Record** or **Union** component definitions use the “(*)” suffix, as for recursive references. For example, the following alternate definition of EntryInformationSelection is acceptable :

```
Record
    {
    AttributesUnion(*) attributeTypes;
    Integer(4) InfoTypes;
    } EntryInformationSelection;

Union
    {
    NULL allAttributes;
    DynamicArray AttributeType select;
    } AttributesUnion;
```

The C typedefs corresponding to these ELROS defintions (again simplified somewhat for pedagogical reasons) would be :

```
typedef     struct EntryInformationSelection
                {
                struct AttributesUnion * attributeTypes;
                int InfoTypes;
                } EntryInformationSelection;

typedef     struct AttributesUnion
                {
                int tag;
                union
                        {
                        NULL allAttributes;
                        struct select
                                {
                                unsigned long len;
```

```
                AttributeType *val;
                } select;
        } un;
} AttributesUnion;
```

So that data structures can be moved within an interface definition without generating a syntactic error, the “(*)” notation also may be used to backward reference a type. As with a forward reference, the ELROS preprocessor translates a type reference postfixed by “(*)” as a **struct** pointer. Backward references are restricted in the same manner as forward references. That is, the type being backward referenced must be either a **Record** or a **Union** and the backward reference may occur only as a component of a **Record** or **Union**.

Generically, forward and backward references using the “(*)” notation or forward references within **DynamicArray** element type definitions, are called *indirect* references. An indirect reference possesses the following characteristics.

When used as the element type of a **DynamicArray**, they are translated as a **struct** pointer rather than as a pointer to the referenced type. When used as the component of a **Record** or **Union**, they are translated as a **struct** pointer rather than as a type expanded inline. Indirect references in **Record** or **Union** fields are never treated as arrays. That is, the ELROS generated code will send or receive one element of the type when the pointer is non-NULL and will not send any value at all when the pointer is NULL.

The ELROS preprocessor generates free routines for all indirect references that are **freeable**. Indirect references may be annotated in the same way as normal references to control free routine generation (see section 2.9.5).

2.9.4 The behavior of string type variables in **give** and **accept** statements

In most scientific work the use of definite length string types, i.e., those that specify an integer value in parentheses after the base type name, is sufficient when writing distributed applications. However, when programming applications that implement ISO protocol specifications (e.g., X.500), there are situations in which the programmer must use the indefinite length form. Indefinite length string variables, i.e., those whose type is one of the indefinite length string types, behave somewhat differently in **give** and **accept** statements than their definite length cognates[15].

When a definite length **BitString**, **OctetString**, **ObjectIdentifier**, or LenTerm character string variable is given as the parameter to an operation in the on-clause of a **give** statement, the ELROS generated code transmits as many bits, octets, components, or characters as the "len" field indicates. If the variable is one of the 4 NULTerm character string types, the ELROS generated code transmits characters from the string until a NUL character is encountered. If a NUL character is not encountered in processing the n+1 characters of the array, an exception is generated.

When using an indefinite length variable as the parameter to an operation in the on-clause of a **give** statement, the programmer must ensure that the relevant pointer references memory holding the variable's value. This can be either the address of a local variable or the address of memory allocated from the heap. Furthermore, the "len" field of a **BitString**, **OctetString**, **ObjectIdentifier** or LenTerm character string variable must be set to the number of elements to be sent.

When a definite length **BitString**, **OctetString**, **ObjectIdentifier** or LenTerm character string variable is given as the parameter to an operation in the on-clause of an **accept** statement, the ELROS generated code stores the bits, octets, components, or characters received into the variable's "val" array and sets the "len" field to the number received. If more than bits, octets, components or characters are received, an exception is generated. If the variable is one of the NULTerm character string types, the ELROS generated code stores characters into the variable until either

[15] While we concentrate on the **give** and **accept** statments here, these comments also apply to the **encode** and **decode** statements described in section V.

all of the received characters are stored, at which point the generated code terminates the string with a NUL character, or the number of characters exceeds the number given in the type declaration. In the latter case, the ELROS generated code raises an exception.

When the programmer uses an indefinite length variable (either a **BitString**, **OctetString**, **ObjectIdentifier** or any character string type) as the parameter to an operation in the on-clause of an **accept** statement, the ELROS generated code allocates space from the heap to hold the value received and updates the variable's pointer to reference the allocated space. Consequently, any value previously in the variable's pointer component is lost after the **accept** statement completes. If the variable is a NULTerm character string type, the generated code stores a NUL character at the end of the received value.

2.9.5 Freeing space allocated within recursive data structures and indefinite string types

As described in section 2.4.1, the ELROS preprocessor generates routines that free any space allocated either by the programmer or by the ELROS generated code and referenced by the "val" field of a DynamicArray. These routines are generated for each DynamicArray type as well as for all types containing one or more DynamicArrays.

The ELROS preprocessor also generates free routines for recursive types, for indefinite length string types and for types containing these types. These routines are accessed through the rofreei() and rofree() routines described in section 2.4.1.

For example, suppose a program declares a variable of type ListResult, allocates space for all embedded components of type ListResult (i.e., those in the unsignedListResult components), and then uses the variable. After its last use the programmer should write a call to rofreei() to free the space referenced within its parts. That is, the following call should be made (assuming the ListResult variable is called "list_result" and the ListResult type occurs in the interface "X_500_DAP_interface") :

```
rofreei("ListResult", &list_result, X_500_DAP_interface);
```

This call will free all allocated space associated with the variable list_result, including all space recursively referenced by the "uncorrelatedListInfo" components as well as any other DynamicArrays that contain allocated space (e.g., the "subordinates" component), also recursively.

A free routine will attempt to free all DynamicArrays, indefinite length strings and recursive data structures contained within the variable specified. However, there are situations when this can cause problems. For example, if the programmer has stored the address of a local variable (i.e., one allocated on the stack, not on the heap) in the pointer component of a freeable data structure, the free routine will attempt to return that space to the heap by calling "free()". This will result in a system error that typically terminates the program.

Normally, ELROS generates a free routine for all types that are or contain DynamicArrays, indefinite length strings and recursive data structures. However, the programmer can annotate data structures so that a free routine will not be generated. This technique is covered in detail in section 3.6.

Free routines called through rofree() and rofreei() cannot be used to free space referenced through user specified C pointers. For example, suppose the programmer declares :

```
ListResult *list_result_pointer;
```

Since this variable is a C pointer to a ListResult type value, the programmer must allocate the space for this value by calling malloc(), e.g.,

```
list_result_pointer = (ListResult *) malloc(sizeof(ListResult));
```

A subsequent call to rofreei() does not free this space. That is, the following call leaves the space referenced by "list_result_pointer" allocated, while freeing all space referenced "within" the ListResult value :

```
rofreei("ListResult", list_result_pointer, X_500_DAP_interface);
```

III. Controlling ASN.1 Encoding Rules

The mapping of ELROS data structures into ASN.1 uses default encoding rules unless these defaults are overridden by the programmer. This is accomplished by annotating the data structure definitions in an ELROS interface. The annotations allow complete control over the ASN.1 generated by the embedded code.

ELROS annotations are always enclosed in angle brackets. They are inserted at various points in the data structure definition, usually following a reserved word, and are treated as part of the typing information. Consequently, two ELROS data structure definitions that are identical except for their encoding annotations are not type equivalent.

3.1 Controlling the tag of an ELROS type

Any of the basic types may be annotated with tag values that override their default encoding. The annotations include the following possibilities :

	universal *integer* **implicit**
application *integer*	**application** *integer* **implicit**
context *integer*	**context** *integer* **implicit**
private *integer*	**private** *integer* **implicit**

Tag value annotations are given in angle brackets following the name of the basic type. The *integer* following the tag class specifies the number of the tag.

For example, there are two useful type definitions given in the ASN.1 specification that are not ELROS basic types : 1) Generalized Time and 2) Universal Time. Using the annotation feature of ELROS these definitions are made as follows :

VisibleString(16) <universal 24 implicit> GeneralizedTime

VisibleString(16) <universal 23 implicit> UTCTime

These specifications, when included in an interface definition, define two types, GeneralizedTime and UTCTime, that can be used subsequently either as elements in further data structure definitions or to define variables of these types in embedded language programs.

The programmer should use **universal implicit** with great care to ensure the compatibility of ELROS data structure encoding with other ASN.1 implementations. In particular, universal class numbers in the range 1-27 should not be used since they are the standard values used for the basic ASN.1 types.

ASN.1 allows type definitions to have a string of tag values preceding their length and value fields. We support these types in ELROS by allowing a string of class definitions to appear in angle brackets.

For example, suppose we needed a private type, call it TimeStamp, that mapped directly to UTCTime. It could be defined in two ways :

UTCTime <private 1> TimeStamp

or

VisibleString(16) <private 1 universal 23 implicit> TimeStamp

The first definition makes use of a UTCTime definition that must appear previously in the interface specification. The second definition defines the TimeStamp type directly from the predefined ELROS types.

The definition of TimeStamp given above results in an encoding consisting of two tags, "private 1" and "universal 23 implicit". Either of the two following definitions eliminates the universal 23 tag in the type encoding :

UTCTime <private 1 implicit> TimeStamp

or

VisibleString(16) <private 1 implicit universal 23 implicit> TimeStamp

The second definition, while legal, is exactly equivalent to :

VisibleString(16) <private 1 implicit> TimeStamp

A series of contiguous implicit tags collapses to the first. Although we show only two tags in this example, ELROS places no restriction on the number of tags that may appear in the string inside the annotating angle brackets.

3.2 Controlling Record element tags

Programmers control Record element tag encodings as they control the tag encodings of ELROS types. Following the type of a Record element an optional angle bracket annotation specifies its encoding. For example, consider the ReadArgumentType data structure defined in section 2.6. Each element of the Record requires a context specific tag to discriminate it from the other elements. Using the ASN.1 definition shown in the same section (as well as in Appendix A), the properly annotated ELROS data structure would be :

```
Record <noautotag>
    {
    Name <context 0> object;
    EntryInformationSelection <context 1> selection;
    ServiceControls <context 30> serviceControls;
    SecurityParameters <context 29> securityParameters;
    DistinguishedName <context 28> requestor;
    OperationProgress <context 27> operationProgress;
    Integer(4) <context 26> aliasedRDNs;
    DynamicArray <context 25> Extension extensions;
    } ReadArgumentType;
```

When a ReadArgumentType variable is encoded and decoded, the context specific tags given in the type definition control the ASN.1 generated and parsed by the ELROS runtime system.

There is another aspect of this example that requires discussion, i.e., the **noautotag** annotation that appears after the reserved word **Record**. By default ELROS tags each Record element with a context specific tag so that it is guaranteed that each element type is unique. These tags begin at 0 and proceed sequentially by 1. That is, the first context specific tag is 0, the next is 1, the next 2, and so forth.

Using autotagging simplifies the ELROS preprocessor generated code, but interferes with manually tagging Record elements. In particular, unless the **noautotag** annotation is used after the **Record** reserved word, the ELROS preprocessor will generate an error if element tagging is specified[16].

Since ASN.1 specifications normally give context specific tags that uniquely discriminate between SET elements, ISO protocol implementations generally will always require noautotagging. For the convenience of ISO protocol implementers, ELROS allows the programmer to chose the default (i.e., either **noautotag** or **autotag**) when an interface is specified. This default applies for all types defined within the interface specification. The default selection is an optional field on the interface identification statement. If it is not present, the default is **autotag**. The syntax of the interface identification statement is :

interface *interface-name* [autotag | noautotag];

The square brackets are not part of the identification line, rather they show that the default selection reserved words are optional.

Since the topic of this tutorial is programming ISO protocols with ELROS, we assume the programmer has chosen noautotagging in the interface definition statement in all subsequent definitions of ELROS data structures (except briefly, when we discuss Unions).

3.3 Optional and default values of Record elements

ASN.1 supports SET and SEQUENCE OPTIONAL elements as well as DEFAULT values. When an element is optional, it need not be sent to the

---

[16] The reader may wonder why we chose autotagging as the default when it would seem that noautotagging would be a better choice. Our decision derives from the fact that those who funded the development of ELROS are interested in scientific rather than system programming. Scientific programmers have little or no interest in how the data is encoded (unless the data encoding strategy affects performance). They are much more interested in the convenience of the programming interface. Since scientific programmers want records to be properly sent and received without thinking about the uniqueness of their element types, we chose autotagging as the default.

correspondent. If an element has a default value, it must not be sent whenever its value is the default. When a correspondent receives a SET or SEQUENCE missing an element with a default value, the ELROS generated code sets the corresponding data structure entry to the default.

ELROS supports optional elements and default values with data structure annotations. Within the optional angle brackets that follow the element type, the programmer may indicate two properties in addition to the tag encoding information. The reserved word **null** is used to specify values of the element that should not be sent to the correspondent. This is achieved in one of two ways.

If the element is one of the basic ASN.1 types, a constant value of that type may follow the reserved word **null**. This annotation directs the ELROS runtime system to compare the value of the element with the constant before it is encoded. If they are equal, the element is not sent to the correspondent. The grammar that specifies how to form a constant of an ELROS basic type is given in appendix C.

For example, the aliasedRDNs element of ReadArgumentType is optional, according to its formal ASN.1 definition. Suppose the programmer decides not to send this element, which is of type Integer, whenever it equals 0. The annotated Record definition that accomplishes this is :

```
Record
    {
    Name <context 0> object;
    EntryInformationSelection <context 1> selection;
    ServiceControls <context 30> serviceControls;
    SecurityParameters <context 29> securityParameters;
    DistinguishedName <context 28> requestor;
    OperationProgress <context 27> operationProgress;
    Integer(4) <context 26 null 0> aliasedRDNs;
    DynamicArray <context 25> Extension extensions;
    } ReadArgumentType;
```

The second method of indicating values of an element that should not be sent to a correspondent uses the reserved word **null** without following it with a constant. The ELROS preprocessor generates a flag element within the corresponding C struct typedef that controls whether the element is to be sent. This flag is named _ _*element-name*_absent, where *element-name* is the name of the element. Below we refer to this as the “absent flag” for the element.

For example, the "requestor" element of ReadArgumentType is optional, according to its ASN.1 definition. However, since it is not an ELROS basic type[17], there is no way to express a constant value that indicates when it should not be sent. Using the **null** reserved word without a constant value in the element annotations directs the ELROS preprocessor to generate the element, _ _requestor_absent, in the equivalent C struct typedef. In particular the specification :

```
Record
    {
    Name <context 0> object;
    EntryInformationSelection <context 1> selection;
    ServiceControls <context 30> serviceControls;
    SecurityParameters <context 29> securityParameters;
    DistinguishedName <context 28 null> requestor;
    OperationProgress <context 27> operationProgress;
    Integer(4) <context 26 null 0> aliasedRDNs;
    DynamicArray <context 25> Extension extensions;
    } ReadArgumentType;
```

generates the typedef[18] :

```
struct
    {
    Name object;
    EntryInformationSelection selection;
```

17 Eventhough the example shown here is for a non-basic ELROS type, this method is applicable to any type including the basic ones.

18 For pedagogical clarity, we have not shown the ELROS types expanded to their corresponding C definitions.

```
ServiceControls serviceControls;
SecurityParameters securityParameters;
DistinguishedName requestor;
int _ _requestor_absent;
OperationProgress operationProgress;
Integer(4) aliasedRDNs;
DynamicArray Extension extensions;
} ReadArgumentType;
```

When the field _ _requestor_absent equals 0, "requestor" is sent to the correspondent. When _ _requestor_absent equals 1, "requestor" is not sent.

ELROS also supports a **default** annotation that gives the value a receiver should use if a Record element is not received. This value is expressed by the reserved word **default** followed by a constant of one of the basic types (see Appendix C for a grammar of these constants). For example, the X.500 element EntryInformationSelection maps into the following ELROS data structure definition :

```
Record
    {
    Union
        {
        NULL allAttributes;
        DynamicArray AttributeType select;
        } attributeTypes;
    Integer(4) InfoTypes;
    } EntryInformationSelection;
```

The Record element InfoTypes defaults to 1. The annotation specifying this default is :

```
Record
    {
    Union
        {
        NULL allAttributes;
```

```
            DynamicArray AttributeType select;
            } attributeTypes;
    Integer(4) <null default 1> InfoTypes;
    } EntryInformationSelection;
```

Note that the annotation specifies not only a **default** value, but also a **null** indication (in this case a **null** generating an “absent flag” for the element). ELROS requires a Record element with a **default** to be annotated with either of the two kinds of **null**. This is necessary so the ELROS runtime system of the sending correspondent can determine when to send the element value. Whenever the receiving correspondent detects the absence of a Record element, it sets the corresponding C struct element to the default value.

Notice that the programmer cannot specify a default value unless the element is one of the basic types. Elements that are not a basic type that have default values must be handled differently. In particular the programmer must initialize the element to the default value before it is used in an accept statement. If the ELROS runtime system does not receive a value for that Record element, it does not disturb the value that is already there. Consequently, a Record element treated in this way retains the default value to which it was set prior to its use in the accept statement. Furthermore, if an element has the **null** annotation and no value is received for it when it is used in an **accept** statement, its absent flag is set to 1.

For example, the ReadArgumentType element "operation_progress" defaults to a singleton set consisting of the enumerated value "notStarted". The ASN.1 definition for the type OperationProgress is :

```
OperationProgress          ::=     SET {
                                        nameResolutionPhase [0]
                                            ENUMERATED {
                                                notStarted (1),
                                                proceeding (2),
                                                completed (3) },
                                        nextRDNToBeResolved [1]
                                            INTEGER OPTIONAL }
```

This maps to the ELROS Record :

```
Record
    {
    Enumerated(1) nameResolutionPhase;
    Integer(1) nextRDNToBeResolved;
    } OperationProgress;
```

In order to create a default value of this Record, the nextRDNToBeResolved element must have a corresponding absent flag. Thus, the annotated version of this Record is :

```
Record
    {
    Enumerated(1) <context 0> nameResolutionPhase;
    Integer(1) <context 1 null> nextRDNToBeResolved;
    } OperationProgress;
```

In the operation_progress element of ReadArgumentType, the programmer initializes the absent flag of nextRDNToBeResolved to 1 (indicating it is absent) and the value of nameResolutionPhase to 1 (which is the integer equivalent of notStarted). If the operation_progress element of ReadArgumentType is not received, this default value and the value of the absent flag is undisturbed by the ELROS generated code. If the operation_progress element is received, the absent_flag will be set to 0 by the ELROS generated code and the value of nameResolutionPhase will be set to the value transmitted by the correspondent.

As another example, the ReadArgumentType elements "selection", "service_controls", and "security_parameters" all default to the empty set. This situation is somewhat easier to deal with. Each of these elements are annotated with **null**, generating an absent flag. The Record definition then appears as :

```
Record
    {
    Name <context 0> object;
```

```
        EntryInformationSelection <context 1 null> selection;
        ServiceControls <context 30 null> serviceControls;
        SecurityParameters <context 29 null> securityParameters;
        DistinguishedName <context 28 null> requestor;
        OperationProgress <context 27> operationProgress;
        Integer(4) <context 26 null 0> aliasedRDNs;
        DynamicArray <context 25> Extension extensions;
        } ReadArgumentType;
```

When a ReadArgumentType variable is used in an **accept** statement, the program can test the absent flags "_ _selection_absent", "_ _service_controls_absent" and "_ _security_parameters_absent" to determine whether those elements were sent. If the absent flag equals 1, they were not.

3.4 Controlling Union element tags

The encoding of Union element tags is controlled in the same way as Record element tag encoding. Tag modifiers are given in angle brackets after the Union element type. For example, the ELROS data structure definition of EntryInformationSelection is :

```
    Record
        {
        Union
              {
               Null allAttributes;
               DynamicArray AttributeType select;
               } attributeTypes;
        Integer(4) <null default 1> InfoTypes;
        } EntryInformationSelection;
```

According to the ASN.1 specification, the Union elements "allAttributes" and "select" are tagged respectively with context specific tags 0 and 1. Consequently, the proper ELROS definition is (ignoring for the moment that the DynamicArray "select" is improperly encoded as a SEQUENCE OF rather than a SET OF – see section 3.5) :

```
Record
    {
    Union <noautotag>
            {
            Null <context 0> allAttributes;
            DynamicArray <context 1> AttributeType select;
            } attributeTypes;
    Integer(4) <null default 1> InfoTypes;
    } EntryInformationSelection;
```

Note that the context specific tag for "select" annotates the type constructor **DynamicArray**, rather than the type of the elements in the array. This is an example of controlling the tagging of constructed types, a topic we cover in more detail below.

Also note that there is a **noautotag** annotation after the reserved word **Union**. This is required because of the way ELROS treats Unions.

ELROS Unions map into the ASN.1 CHOICE construct. According to the ASN.1 standard, CHOICE is represented by the type of the underlying selected element. For example, if the selected Union element of "attributeTypes" is "select", the element sent has a type of SEQUENCE, the default encoding of a DynamicArray. ASN.1 has no typing information that indicates that SEQUENCE was the selected element of a CHOICE type.

For this reason, all elements of a Union must be distinguishable by their types. If this is not true, typing information must be added so that a receiving correspondent can determine which element of the Union was sent. ELROS supplies the necessary discriminating type information by automatically tagging each Union element with a context specific tag. As with Records, these tags begin at 0 and are incremented by 1 for each subsequent element. If the programmer desires to manually tag Union elements with typing information, he must turn off autotagging by using the **noautotag** annotation (otherwise ELROS will flag the Union element annotations as an error).

As with Record definitions, the tagging default (i.e., either **autotag** or **noautotag**) appearing in the interface identification statement applies to Unions. For the purpose of subsequent discussions, we assume the programmer has selected **noautotag** as the default for all types we define.

As an aside, note that for the particular example we have chosen, autotagging provides the correct result. That is, ELROS will correctly tag the elements of the Union "attributeTypes" with the context specific tags given in the ASN.1 specification. However, relying on such a coincidence is dangerous practice, since changes to the standard might introduce elements with context specific tags that are not sequential, which could become a source of error. In general it is best to always manually tag the elements of a Union in an ISO protocol implementation.

3.5 Controlling the encoding of constructed types

By default a Record is encoded as a SET and an Array or DynamicArray as a SEQUENCE OF. ELROS provides annotations that allow the programmer to override these defaults.

We have already seen that Records and Unions can be annotated to control autotagging. Records and arrays can also be annotated to control the tag of the constructed type. After the reserved words **Record**, **Array**, or **DynamicArray** an angle bracket annotation may appear that contains either the reserved word **set** or **sequence** (but not both). In addition, a series of type specifications drawn from the class definitions given in section 3.1 may also appear within the angle brackets. This series must precede the **set** or **sequence** annotation, if either of these are present.

For example, the ASN.1 specification for ReadArgument is :

```
ReadArgument            ::=     CHOICE {
                                        ReadArgumentType,
                                        SEQUENCE {
                                                ReadArgumentType,
                                                AlgorithmIdentifier,
                                                BIT STRING }  }
```

with the corresponding ELROS data structure definition :

```
Union
    {
    ReadArgumentType  readArgument;
    Record <sequence>
            {
            ReadArgumentType  readArgument;
            AlgorithmIdentifier  algorithmIdentifier;
            BitString(2048) signature;
            } signatureBlock;
    }  ReadArgument;
```

The annotation to the Record "signatureBlock" overrides the default **Record** encoding (i.e., SET), specifying the encoding to be a SEQUENCE.

Another example where the default encoding of a constructed type must be overridden occurs in the type EntryInformationSelection. Its ASN.1 specification is :

```
EntryInformationSelection        ::=      SET {
                                                  attributeTypes CHOICE {
                                                     allAttributes [0] NULL,
                                                     select [1] SET OF AttributeType }
                                                          DEFAULT allAttributes NULL,
                                                  InfoTypes [2] INTEGER
                                                          {attributeTypesOnly (0),
                                                           attributeTypesAndValues (1)}
                                                           DEFAULT
                                                                  attributeTypesAndValues  }
```

which maps to the ELROS data structure definition :

```
Record
    {
    Union
           {
            Null <context 0> allAttributes;
```

```
            DynamicArray <context 1 set> AttributeType select;
            } attributeTypes;
    Integer(4) <null default 1> InfoTypes;
    } EntryInformationSelection;
```

Here the type of the DynamicArray "select" is encoded by a context specific tag of 1 followed by a SET tag (which according to the ASN.1 standard is equivalent to a SET OF tag).

One final word about overriding the default constructed encoding of a **Record**, **Array**, or **DynamicArray**. The reserved words **set** and **sequence** must appear only once in the angle bracket annotation and can appear only at the end of a series of type specifications if present. These reserved words are special in the annotation in that they are not actually part of the type specification series. A good rule to determine how a **Record**, **Array**, or **DynamicArray** is encoded is to first examine the end of the angle bracket annotation to see if a **set** or **sequence** keyword is present. If so, the tag of the constructed type is changed. The series of type specifications is then evaluated to modify the tag. This evaluation must take into account any **implicit** annotations as well as the position of the types in the series. Only after this complete evaluation will the tag structure associated with the constructed type be known.

3.6 Controlling the generation of free routines

ELROS allows a programmer to annotate type definitions to control the generation of free routines. These annotations appear in angle brackets after the **Record**, **Union**, **Array** and **DynamicArray** reserved words. The reserved word **freeable** is used to instruct the preprocessor to generate a free routine and the reserved word **unfreeable** is used to instruct it to inhibit generation. Normally, if a data structure is unannotated, the preprocessor will generate a free routine. However, the programmer is allowed to annotate the interface definition statement to specify that all **Record**, **Union**, **Array** and **DynamicArray** types are **freeable** or **unfreeable**. The programmer can then override the interface default in the definition of particular types.

For example, consider the ReadArgumentType definition from the X_500_DAP_interface. Suppose the programmer wrote the interface as follows :

```
interface X_500_DAP_interface <unfreeable>;
                              .
                              .
                              .
    Record <freeable>
        {
        Name <context 0> object;
        EntryInformationSelection <context 1 null> selection;
        ServiceControls <context 30 null> serviceControls;
        SecurityParameters <context 29 null> securityParameters;
        DistinguishedName <context 28 null> requestor;
        OperationProgress <context 27> operationProgress;
        Integer(4) <context 26 null 0> aliasedRDNs;
        DynamicArray <context 25> Extension extensions;
        } ReadArgumentType;

end;
```

The preprocessor would not generate free routines for types defined in the interface unless the programmer specifically annotates the type definition with the **freeable** reserved word (as is done for ReadArgumentType). This implies that the preprocessor will not generate a free routine for the "extensions" component of ReadArgumentType, since it is unannotated and consequently is controlled by the interface default.

Use of the **freeable** and **unfreeable** reserved words gives the programmer significant control over the memory allocation strategies implemented in executable code. For example, assuming the definition of ReadArgumentType given above, when the ReadArgumentType free routine is called (by the appropriate call to either rofreei() or rofree()), storage referenced by the "extensions" component of that variable is not freed. This allows the programmer the option of explicitly allocating and freeing the space for the "extensions" component from the heap or storing the address of a local variable array of type “Extension” in the “extensions” “val” field.

Even with this flexibility, there may be times when the programmer wishes to use the address of a local variable for a **freeable** data structure or the address of memory from the heap for an **unfreeable** data structure. In the latter case, the executable code must manually free the space allocated, since no free routine is available to do this. In the former case, the programmer must ensure that the type’s free routine is not called, since a system error would result if it were. This can be problematical if the variable is deeply embedded in an encapsulating data structure that has a free routine. When the free routine for the encapsulating data structure is called, it will automatically call the free routine for the variable holding the address of the local variable.

To accommodate this situation, a free routine first tests to determine if the pointer it would pass to free() is NULL. If so, the free routine returns successfully without calling free(). This allows the programmer to explicitly store NULL in the pointers of embedded **freeable** data structures that reference local variables. When the encapsulating free routine is called, the call to the embedded free routine will not attempt to free the space associated with the embedded variable, since the associated pointer is NULL.

IV. ELROS Error Handling

There are two categories of error that can occur in an ELROS program : 1) errors generated by the ELROS embedded language program (called **exceptions**), and 2) application errors causing a correspondent to send an **error** or **reject** operation.

4.1 Exceptions

The experienced distributed application programmer knows that error handling is one of the most troubling aspects of his craft. The error modes that arise in distributed systems include those that occur in stand-alone systems as well as others that are significantly harder to correct.

We have already seen in another context the basic mechanism provided by ELROS for handling ELROS generated exceptions, i.e., the **exceptions** clause. The **exceptions** clause can be attached to the **bind_addr** statement, the **unbind** statement, the **accept** statement, the **give** statement and other statements described in later chapters. We have used the **exceptions** clause to detect the unbinding of an instance using the **done** clause. However, there are other on-clauses that may appear within the **exceptions** clause. They are (in the following definitions square brackets are used to show optional fields; they are not literal text) :

on com_error(*status* [, *string-variable*]) *C-statement*

When a communications error occurs during the execution of the statement to which the **exceptions** clause is attached, control passes to the **com_error** on-clause. An example of a **com_error** is a timeout on a **bind_addr**, **give**, or **accept** statement. The first parameter in parentheses, *status*, is an integer variable which receives the numerical code of the communications error. The second optional parameter, *string-variable*, is a string variable capable of storing at least 512 characters. This variable will receive a textual description of the error. The *C-statement* is executed when the **com_error** on-clause gains control.

on internal_error(*status* [, *string-variable*]) *C-statement*

If an ELROS runtime system failure occurs, control passes to the **internal_error** on-clause. The first parameter in parentheses, *status*, is an integer variable which receives the numerical code of the internal error. The second optional parameter, *string-variable*, is a string variable capable of storing at least 512 characters. This variable will receive a textual description of the error. The *C-statement* is executed when the **internal_error** on-clause gains control.

on protocol_error(*status* [, *string-variable*]) *C-statement*

When an error in the underlying Remote Operations protocol occurs, control passes to the **protocol_error** on-clause. This error indicates that the remote operations service element of the correspondent is faulty. The first parameter in parentheses, *status*, is an integer variable which receives the numerical code of the protocol error. The second optional parameter, *string-variable*, is a string variable capable of storing at least 512 characters. This variable will receive a textual description of the error. The *C-statement* is executed when the **protocol_error** on-clause gains control.

on missing_op(*string-variable*) *C-statement*

When an **accept** statement receives an operation for which no on-clause is specified, control passes to the **missing_op** on-clause. The first parameter, *string-variable*, is a string variable capable of storing at least 32 characters. This variable will receive the operation name, if the operation is defined for the interface. Otherwise, it indicates that the operation is unknown in the interface. The *C-statement* executed when the **missing_op** on-clause gains control.

on missing_arg(*op-string-variable*, *arg-string-variable*) *C-statement*

When an **accept** statement receives an operation argument for which no parameter is specified, control passes to the **missing_arg** on-clause. The first parameter, *op-string-variable*, is a string variable capable of storing at least 32 characters. This variable will receive the operation name. The second parameter, *arg-string-variable*, receives the identifier of the argument, which for ISO application protocol implementations will always equal 1. The *C-statement* is executed when the **missing_arg** on-clause gains control.

When more than one on-clause is present inside an **exceptions** clause, an exception is directed to the appropriate on-clause. When the on-clause completes, control passes outside the **exceptions** clause. This flow of control is different from that provided by a **switch** statement, which continues execution of statements below a **case** label until either a **break** statement or the end of the **switch** statement is reached.

As an example of how to use an **exceptions** clause to catch and handle ELROS generated errors, consider the program fragment we give in section 2.8.3 corresponding to an X.500 performer. Exceptions can be generated by the **bind_addr** statement, by the **accept** statement and its embedded **give** statement, and by the **unbind** statement. Attaching an **exceptions** clause to each of these statements and including the appropriate on-clauses in them results in the following code :

```
                    .
                    .
                    .
/* establish communications with the invoker */

    bind_addr(X_500_invoker, asn_1_ber, &addr, 10);
    exceptions
        {
        on com_error(status, text)
                com_error_handler(status, text);
        on internal_error(status, text)
                internal_error_handler(status, text);
        on protocol_error(status, text)
```

```
                protocol_error_handler(status, text);
        }

/* obtain the ReadOp requests and service them. When the invoker unbinds,
    unbind the communications instance and exit the while loop */

    bound = TRUE;

    while(bound)
        {
        accept X_500_invoker
            {
            on ReadOp(read_argument)
                {
                /* call the subroutine that performs a read on the X_500
                    database */

                read_database(&read_result, &read_argument);

                /* return the read result to the invoker */

                give X_500_invoker
                    ReadResultOp(read_result);
                exceptions /* give statement exceptions */
                    {
                    on com_error(status, text)
                        bound = com_error_handler(status, text);
                    on internal_error(status, text)
                        bound = internal_error_handler(status, text);
                    on protocol_error(status, text)
                        bound = protocol_error_handler(status, text);
                    }

                }
            }
        exceptions /* accept statement exceptions */
```

```
        {
        on com_error(status, text)
             bound = com_error_handler(status, text);
        on internal_error(status, text)
             bound = internal_error_handler(status, text);
        on protocol_error(status, text)
             bound = protocol_error_handler(status, text);
        on missing_op(text)
             bound = missing_op_handler(text);
        on done
             {
             unbind(X_500_invoker);
             bound = FALSE;
             }
        }
    }

/* unbind the communications instance */

   unbind(X_500_invoker);
   exceptions
     {
     on com_error(status, text)
            com_error_handler(status, text);
     on internal_error(status, text)
            internal_error_handler(status, text);
     on protocol_error(status, text)
            protocol_error_handler(status, text);
     }
```

The *C-statement* following each on-clause may provide any semantics the programmer chooses, e.g., closing or flushing open files, printing explanatory messages to a control terminal or log file, and perhaps terminating the process. In the example above, these semantics would be implemented in the procedures with names ending in "_handler". These procedures should return FALSE, so that the performer will exit its **while** loop when an exception occurs.

When an exception other than **done** occurs, the state of the communications instance becomes indeterminate. The programmer should not execute any ELROS statement (including **unbind)** that uses an instance on which an exception other than **done** occurs.

When the ELROS preprocessor encounters a statement that takes an **exceptions** clause, but has none provided by the programmer, it attaches a default **exceptions** clause to it. The default clause contains all legal on-clauses for the statement. These on-clauses catch the exception and terminate the process. The ELROS preprocessor also provides default on-clauses for an **exceptions** clause provided by the programmer, which does not specify all possible exceptions. These default on-clauses also terminate the process. Consequently, if the programmer always wishes the process to terminate when an error exception occurs, he need supply only those **exceptions** clauses that handle the **done** condition.

4.2 Rejects

In chapter 2 we stated that ELROS operations may belong to any one of the remote operations classes – **invoke**, **result**, **error** and **reject**. We describe the definition and use of **invoke**, **result** and **error** operations in chapter 2. We now cover the definition and use of **reject** operations.

A **reject** operation is defined in an interface specification with the following syntax :

reject *reject-name reject-problem reject-value* ;

reject-name is a character string that uniquely identifies the **reject** operation. It corresponds to the *operation-name* of **invoke**, **result** and **error** operations. *reject-problem* is an integer value in the range 0-3 specifying the type of reject. These problems are : 0) GeneralProblem, 1) InvokeProblem, 2) ReturnResultProblem, and 3) ReturnErrorProblem. These problem classes are described in ISO 9072-2, the remote operations protocol specification. *reject-value* is an integer specifying the rejection's value. The meaning of these values are also found in ISO 9072-2.

ELROS only allows the definition of user rejects. The ELROS runtime system may generate a provider reject, which is captured and handled by the correspondent's runtime system, but there is no way for an applications programmer to define a provider reject.

Reject operations are unique in that they take no parameter. They are distinguished by their *reject-problem* and *reject-value*, which are constants in the reject operation definition. Consequently, when **reject** operations are used in **give** and **accept** statements, no parameters are specified. Instead, the **reject** operation itself indicates exactly what problem occurred.

V. Encode and Decode - Dealing Directly With ASN.1

ELROS not only supports the sending and receiving of remote operations, it also allows the applications programmer to send and receive ASN.1 encoded data not formatted according to the remote operations standard. This capability is provided through the **encode** and **decode** statements and the **data** instance declarator.

5.1 ASN.1 Communications Instances

In order to properly handle the encoding and decoding of user data, ELROS must know whether the programmer intends to use the remote operations standard for a particular communications instance or to encode/decode instance data directly into ASN.1. The programmer communicates his intentions through the use of the instance declaration. We have already described the use of the **control** reserved word to declare an instance variable that is used for remote operations. The reserved word **data** is used to declare an instance variable that is used for direct ASN.1 encoding. The syntax for a **data** instance declaration is very similar to the **control** instance declaration :

data (*interface-name*) *instance-variable-list*;

Notice that the **data** instance declaration also takes the name of an interface in parentheses. This ties all instance variables declared in the *instance-variable-list* to that interface. The only data structures that can be sent or received over instances represented by these variables are those found in the specification of *interface-name*.

For example, suppose we want to send a ReadArgument value to a correspondent without placing it in a remote operation[19]. This requires an

[19] This example is somewhat artificial. It is probably better programming practice to define an remote operation that takes a ReadArgument parameter and use it for communicating such values between correspondents. The main justification for **encode** and **decode** is its use in the Any macros described below. However, this use is hidden from the user. The only other utility of **encode** and **decode** is to

instance tied to the X_500_DAP_interface. The appropriate instance declaration would be :

data(X_500_DAP_interface) read_arg_inst;

Once the instance variable is declared, it must be bound to a correspondent by calling **bind_addr**.

5.2 Encoding ASN.1 Directly

Once a data instance is declared and bound to a correspondent, it can be used in **encode** and **decode** statements. An **encode** statement sends ASN.1 encoded data to a correspondent, while a **decode** statement receives ASN.1 encoded data.

The **encode** statement begins with the reserved word **encode**, followed by an *instance-variable* and then by a C language statement. The exact syntax is :

encode *instance-variable C-statement*

Within the *C-statement* the programmer embeds encoding clauses of the following form :

expression => *interface-type*

The value of *expression* is encoded in the ASN.1 representation that corresponds to *interface-type*. *interface-type* must be defined in the interface specification to which *instance-variable* was tied when it was declared. Normally the type of *expression* is *interface-type*.

Each **encode** statement may contain any number of encoding clauses as long as they are placed within a C compound statement. They may appear

---

implement non-ISO standard protocols that are represented directly in ASN.1. Examples are SNMP and Kerberos. However, ELROS is unsuitable for building SNMP implementations, since ELROS uses indefinite length encoding, which is prohibited by that protocol.

within loops or any other C control structure. As with operation references within **give** statements, each time an encoding clause is executed within an **encode** statement, a new PDV is created and placed within the PPDU. The PPDU is sent only after control passes from the **encode** statement.

For example, suppose we have defined an array of ReadArguments and wish to send them to a correspondent. The following code accomplishes this :

```
/* import the X_500 interface */

#include  "~/dist/interfaces/X_500_DAP_fragment.i"

/* declare the instance, address structure variable, control variable and
    read_argument  array  */

    data(X_500_DAP_interface)  read_arg_inst;
    bind_address  addr;
    int i;
    ReadArgument  read_argument[20];

/* fill in the values of the read_argument array */
                                .
                                .
                                .
/* initialize the address structure and fill in the correspondents address */

    init_isode_addr(&addr, initiator, "Internet=134.8.20.12+2046");

/* establish communications with the performer */

    bind_addr(read_arg_inst, asn_1_ber, &addr, 10);
    exceptions
        {
        on com_error(status, text)
                com_error_handler(status,  text);
        on internal_error(status, text)
```

```
                internal_error_handler(status, text);
        on protocol_error(status, text)
                protocol_error_handler(status, text);
        }

/* send the 20 read arguments to the correspondent */

    for(i=1, done=0; i<20 && !done; i++)
        {
        encode  read_arg_inst
            {
            read_argument[i]  =>  ReadArgument;
            }
        exceptions
            {
            on com_error(status, text)
                  {
                  com_error_handler(status, text);
                  done=1;
                  }
            on internal_error(status, text)
                  {
                  internal_error_handler(status, text);
                  done=1;
                  }
            on protocol_error(status, text)
                  {
                  protocol_error_handler(status, text);
                  done=1;
                  }
            }
        }

/* unbind the communications instance */

    unbind(read_arg_inst);
```

```
exceptions
    {
    on com_error(status, text)
            com_error_handler(status, text);
    on internal_error(status, text)
            internal_error_handler(status, text);
    on protocol_error(status, text)
            protocol_error_handler(status, text);
    }
```

There are two notable things about this example. First, the interface type ReadArgument is used to declare an array of variables by using normal C syntax. That is, there is no definition of an array of ReadArguments in the X_500_DAP_interface specification. The use of normal C syntax to construct new types can sometimes be useful. Of course, types constructed in this way cannot be sent using ELROS. Only types defined in an interface specification are communicable over an instance.

Secondly, the **encode** statement is shown with an **exceptions** clause. Both **encode** and **decode** statements may attach an **exceptions** clause to handle ELROS generated errors.

5.3 Decoding ASN.1 Directly

The ELROS **decode** statement is used to receive data directly encoded in ASN.1. Its syntax is very similar to that of an **accept** statement :

decode *instance-variable*

{

on *interface-type*$_1$ => (*variable*$_1$)

C-statement$_1$

on *interface-type*$_2$ => (*variable*$_2$)

C-statement$_2$

.

.

.

on *interface-type*$_n$ => (*variable*$_n$)

C-statement$_n$
}

Each on-clause embedded in a **decode** statement specifies one type from the interface to which *instance-variable* is tied, followed by a parenthesised *variable*[20] that will receive the value if the correspondent sends that type. The *C-statement* at the end of each on-clause performs any actions the programmer decides are necessary when a value of the indicated type is received.

If a **decode** statement has multiple on-clauses, the encoding of each type appearing in these on-clauses must be different at the outer most level. For example, if two on-clauses specify Integer types, they must be tagged in such a way that their encoding tag values are different. This is also true of constructed types represented as SETs, SEQUENCEs or UNIONs. To illustrate this point, consider the following ELROS code :

```
interface example;

Integer(2) int1;
Integer(4) int2;
Integer(8) int3 <private 1>;

Record
    {
    Integer(2) i;
    Integer(4) j:
    } rec1;


Record
    {
    Boolean(1) bool1;
    Boolean(1) bool2:
    } rec2;
```

[20] Any lvaule of the appropriate type may be used in place of *variable*.

```
Record <sequence>
    {
    Integer(2) i;
    Integer(4) j:
    } rec3;

end;
```

Neither of the following **decode** statements are legal :

decode inst
    {
    on int1 => (i)
        C-statement$_1$
    on int2 => (j)
        C-statement$_2$
    }

decode inst
    {
    on rec1 => (ints)
        C-statement$_1$
    on rec2 => (bools)
        C-statement$_2$
    }

Both int1 and int2 are encoded as base Integer types, while both rec1 and rec2 are encoded as SETs.

However, the following **decode** statements are legal :

decode inst
    {
    on int1 => (i)
        C-statement$_1$

```
    on int3 => (j)
        C-statement3
    }

decode inst
    {
    on rec1 => (ints1)
        C-statement1
    on rec3 => (ints3)
        C-statement3
    }
```

They are legal since int1 is tagged as an Integer, while int3 is tagged as a "private 1". Similarly, rec1 is tagged as a SET, while rec3 is tagged as a SEQUENCE.

Like the **accept** statement, the **decode** statement does not loop. Consequently, if more than one value is expected, the **decode** statement must be embedded within a C statement that iterates.

Both the **encode** and **decode** statements take an **exceptions** clause. The **done**, **com_error**, **internal_error** and **protocol_error** on-clauses may appear within the **exceptions** clause as well as the **missing_arg** on-clause, which gains control when a type not specified in a **decode** statement arrives over the instance.

To illustrate how the **decode** statement might be used, consider the following program fragment that receives the 20 ReadArgument values sent by the correspondent described in the previous section :

```
/* import the X_500 interface */

#include  "~/dist/interfaces/X_500_DAP_fragment.i"

/* declare the instance, address structure variable, control variable and
    operation parameters */

    data(X_500_DAP_interface)  read_arg_inst;
    bind_address  addr;
```

```
    int i;
    ReadArgument  read_argument[20];

/* initialize the address structure and fill in the performer's address */

    init_isode_addr(&addr, responder, "Internet=134.8.20.12+2046");

/* establish communications with the invoker */

    bind_addr(read_arg_inst, asn_1_ber, &addr, 10);
    exceptions
        {
        on com_error(status, text)
                com_error_handler(status, text);
        on internal_error(status, text)
                internal_error_handler(status, text);
        on protocol_error(status, text)
                protocol_error_handler(status, text);
        }

/* obtain the ReadArgument values. When the correspondent unbinds,
    unbind the communications instance and exit the while loop */

    for(i=1, done=0; i<20 && !done; i++)
        {
        decode  read_arg_inst
              {
              on ReadArgument => read_argument[i]
                    {

                    /* consume the value in read_argument[i] */
                                        .
                                        .
                                        .
                    }
              }
```

```
        exceptions
            {
            on com_error(status, text)
                  {
                  com_error_handler(status, text);
                  done=1;
                  }
            on internal_error(status, text)
                  {
                  internal_error_handler(status, text);
                  done=1;
                  }
            on protocol_error(status, text)
                  {
                  protocol_error_handler(status, text);
                  done=1;
                  }
              on done
                    {
                    unbind(read_arg_inst);
                  done=1;
                    }
              }
        }

/* unbind the communications instance */

    unbind(read_arg_inst);
    exceptions
        {
        on com_error(status, text)
                com_error_handler(status, text);
        on internal_error(status, text)
                internal_error_handler(status, text);
        on protocol_error(status, text)
                protocol_error_handler(status, text);
```

}

5.4 The Any Type

Each of the ELROS basic types, except **Any**, is well defined in the sense that the structure of the ASN.1 representing them is known at compile-time. The **Any** type, on the other hand, may hold a value of arbitrary ASN.1 structure.

When the ELROS runtime system sends a variable of type **Any** to a correspondent (inside either a **give** or **encode** statement), it assumes the value of the variable is an appropriate ASN.1 encoding. Similarly, when a variable of type **Any** is received from a correspondent (inside either an **accept** or **decode** statement), the ELROS runtime system moves the ASN.1 encoding that represents it into the variable supplied by the application programmer.

To manipulate variables of type **Any**, the programmer uses special macros that move the value of a variable to/from its ASN.1 encoding. These macros are implemented using the ELROS **encode** and **decode** statements.

The macro **encodeAny** encodes a value into its ASN.1 representation. It has the following form :

encodeAny(*interface*, *type_name*, *Any_variable*, *expression*)

Its four arguments are : 1) an interface name, 2) the name of a type defined within that interface, 3) a variable of type **Any** that will receive the ASN.1 encoding, and 4) an *expression*, which must evaluate to a value of type *type_name*.

The macro **decodeAny** decodes the ASN.1 into a variable of type **Any**. It has the following syntax :

decodeAny(*interface*, *type_name*, *variable*, *Any_expression*)

Its four arguments are : 1) an interface name, 2) the name of a type defined within that interface, 3) a *variable*[21] of that type, and 4) an expression of type **Any** that holds the ASN.1 to be decoded.

Two other macros support the encoding and decoding of **Any** values. The **copyAny** macro copies the value of one **Any** variable into another. It has the following syntax :

copyAny(*destination*, *source*)

The value in the variable *source* is copied to the variable *destination*. Both variables must be of type **Any**.

Note that two variables of type **Any** can be used in an assignment statement. For example, if X and Y are two **Any** variables, the statement X = Y is perfectly legal. However, it has a different effect than using the **copyAny** macro, i.e., copyAny(X, Y). The assignment statement *aliases* the variables X and Y. That is, when the value of X is changed, the value of Y changes as a side effect and vice versa. The **copyAny** macro copies the value of Y into X without creating the aliasing problem. When the **copyAny** macro is used, subsequent changes to either X or Y do not change the value in the other variable. Since aliasing can be the source of subtle program bugs, the ELROS programmer is advised to use **copyAny** rather than an assignment statement when copying the value of **Any** variables.

The **clearAny** macro is used for storage management. In particular, when an **Any** variable is used in an **encodeAny** macro, space is allocated from the heap to hold the generated ASN.1. This space must be freed before the **Any** variable itself is freed (if it was created using malloc) or before exiting the block containing its declaration (if it is an automatic variable). The **clearAny** macro executes the appropriate call to the free procedure. It has the following syntax :

clearAny(*Any_variable*)

---

[21] Any lvalue that evaluates to the Any type is legal.

It is good programming practice to immediately call **clearAny** after an **Any** variable has been used for the last time. This prevents memory leaks due to unfreed space allocated by **encodeAny**.

The **Any** macros are implemented with normal ELROS statements that may generate exceptions. Since they are C language macros, there isn't a way to attach an **exceptions** clause to them. However, there is a defined symbol used in the macro bodies in place of the ELROS statement **exceptions** clauses. This allows the programmer to specify an **exceptions** clause that will be attached to each internal ELROS statement. The defined symbol is **AnyExceptions**. For example, the following define statement specifies an **exceptions** clause for all of the **Any** macros :

```
#define AnyExceptions         \
    exceptions                \
        {                     \
        on  com_error         \
            {                 \
            .                 \
            .                 \
            .                 \
        }                     \
```

This **exceptions** clause is used inside all **Any** macros until the symbol **AnyExceptions** is redefined or it is undef'd. The symbol is undefined initially, which causes the ELROS statements within the Any macros to generated the default **exceptions** clause.

As an example of how a programmer might use the **Any** macros, consider an application that sets the "object" element in a ReadArgument variable and makes a ReadOp request of an X.500 performer. The relevant ASN.1 is :

```
    AttributeValue          ::=     ANY

    AttributeType           ::=     OBJECT IDENTIFER

    Attribute               ::=     SEQUENCE {
```

```
                                        type            AttributeType
                                        values          SET OF AttributeValue
                                        }

AttributeValueAssertion         ::=     SEQUENCE {AttributeType,
                                                  AttributeValue}

RelativeDistinguishedName ::=           SET OF AttributeValueAssertion

RDNSequence                     ::=     SEQUENCE OF
                                            RelativeDistinguishedName

Name                            ::=     CHOICE { RDNSequence }

ReadArgumentType                ::=     SET {
                                            object [0] Name,
                                            selection [1] EntryInformationSelection
                                                DEFAULT {},
                                            [30] ServiceControls DEFAULT {},
                                            [29] SecurityParameters DEFAULT {},
                                            requestor [28]
                                                DistinguishedName  OPTIONAL,
                                            [27] OperationProgress
                                                DEFAULT {notStarted}
                                            aliasedRDNs [26]
                                                INTEGER  OPTIONAL,
                                            extensions [25] SET OF Extension
                                                OPTIONAL  }

ReadArgument                    ::=     CHOICE {
                                            ReadArgumentType,
                                            SEQUENCE {
                                                ReadArgumentType,
                                                AlgorithmIdentifier,
                                                BIT STRING }  }
```

Expanding only the relevant portions of ReadArgument into its ELROS equivalent data structures results in the following :

```
Record <sequence>
    {
    ObjectIdentifier(20, 4)  attribute_type;
    Any attribute_value;
    } AttributeValueAssertion;

DynamicArray <set> AttributeValueAssertion RelativeDistinguishedName;

Record
    {
    Union <context 0>
        {
        DynamicArray RelativeDistinguishedName RDNSequence;
        } object;
    EntryInformationSelection selection;
    ServiceControls serviceControls;
    SecurityParameters securityParameters;
    DistinguishedName requestor;
    OperationProgress operationProgress;
    Integer(4) aliasedRDNs;
    DynamicArray Extension extensions;
    } ReadArgumentType;

Union
    {
    ReadArgumentType  readArgument;
    Record
            {
            ReadArgumentType  readArgument;
            AlgorithmIdentifier  algorithmIdentifier;
            BitString(2048) signature;
            } signatureBlock;
    } ReadArgument;
```

Here we have expanded many of the ELROS data structures inline, rather than using separate definitions, which would be the normal approach. We chose inline expansion to achieve compactness in the presentation of the ELROS data structures.

Expanding the ELROS data structure definitions into their equivalent C struct typedefs results in :

```
typedef
    struct
        {
        struct
            {
            int size;
            unsigned long comp[20];
            } attribute_type;
        Any attribute_value;
        } AttributeValueAssertion;

typedef
    struct
            {
            unsigned long len;
            AttributeValueAssertion *val;
            } RelativeDistinguishedName;

typedef
    struct
        {
        struct
            {
            int tag;
            union
                {
                struct
                    {
```

```
                    unsigned long len;
                    RelativeDistinguishedName *val;
                    } RDNSequence;
                } un;
            } object;
        EntryInformationSelection selection;
        ServiceControls serviceControls;
        SecurityParameters securityParameters;
        DistinguishedName requestor;
        OperationProgress operationProgress;
        Integer(4) aliasedRDNs;
        struct
            {
            unsigned long len;
            Extensions *val;
            } extensions;
        } ReadArgumentType;

typedef
    struct
        {
        int tag;
        union
            {
            ReadArgumentType readArgument;
            struct
                {
                ReadArgumentType readArgument;
                AlgorithmIdentifier algorithmIdentifier;
                BitString(2048) signature;
                } signatureBlock;
            } un;
        } ReadArgument;
```

Given these definitions, the following program fragment loads a "common name" attribute into the object element of ReadArgumentType, which itself is an element of ReadArgument. The fragment then sends the read request to the X.500 provider and accepts the ReadResult the provider returns. The fragment iterates over 4 values of "common name".

```
/* import the X_500 interface, declare the instance, address structure
     variable, control variable, and RelativeDistinguishedName pointer. */
                                        .
                                        .
                                        .
/*  Declare the operation parameters, the control variables and a variable to
     receive encodeAny error messages, and declare and initialize the
     common_name array */

     ReadArgument  read_argument;
     ReadResult read_result;
     int i;
     RelativeDistinguishedName *rdn;

     PrintableString(64) common_name[4];

/* Initialize the address structure and fill in the performer's address,
     establish communications with the performer. */
                                        .
                                        .
                                        .
     strcpy(common_name[0], "Danny M. Nessett");
     strcpy(common_name[1], "James A. Guse");
     strcpy(common_name[2], "Michael L. Branstetter");
     strcpy(common_name[3], "Linda C. Stanberry");

/* set the outer Union tag to zero */

     read_argument.un.tag = 0;

/* set the Union tag of "object" to zero */
```

```
    read_argument.un.read_Argument.object.tag = 0;

/* set the number of Relative Distinguished Names to one */

    read_argument.un.read_Argument.object.un.RDNSequence.len = 1;

/* malloc space for the Relative Distinguished Name */

    read_argument.un.read_Argument.object.un.RDNSequence.val =
      (RelativeDistinguishedName *) malloc(sizeof(RelativeDistinguishedName));

/* malloc space for the Attribute Value Assertion within the RDN */

    read_argument.un.read_Argument.object.un.RDNSequence.val[0].val =
      (AttributeValueAssertion *) malloc(sizeof(AttributeValueAssertion));

/* get an intermediate pointer for cleaner looking code */

    rdn = &read_argument.un.read_Argument.object.un.RDNSequence.val[0];

/* set the number of attributes */

    rdn->len = 1;

/* set the number of elements in the ObjectIdentifier */

    rdn->val[0].attribute_type.size = 4;

/* set the ObjectIdentifier element of the attribute_type element (this uses the
    Const macro for ObjectIdentifiers which is covered in chapter 6) */

    Const_ObjectIdentifier(rdn->val[0].attribute_type.comp, 2.5.4.3, 20, 4);

/* In a for loop, read the X.500 entity corresponding to each common name */

    for(i=1, done=0; i<4 && !done; i++)
```

```
{

/* encode the common name into the Any element of attribute_type */

encodeAny(X_500_DAP_interface, PrintableString(64),
                rdn->val[0].attribute_value, common_name[i]);

/* make a read request of the performer */

give X_500_performer
    {
    ReadOp(read_argument);
    }
exceptions /* give statement exceptions */
    {
    on com_error(status, text)
        {
        com_error_handler(status, text);
        done = 1;
        }
    on internal_error(status, text)
        {
        internal_error_handler(status, text);
        done = 1;
        }
    on protocol_error(status, text)
        {
        protocol_error_handler(status, text);
        done = 1;
        }
    }

/* Reclaim the memory allocated by encodeAny */

    clearAny(rdn->val[0].attribute_value);
```

```
        /* Wait for the result and process it */
                        .
                        .
                        .
        } /* end of for loop */

/* free the space allocated for the Relative Distringuished Name */

    rofreei("ReadArgument", &read_argument, X_500_DAP_interface);
```

The performer program fragment accepts a ReadOp request and then calls its read_database subroutine, giving the read_argument as a parameter. Within the read_database procedure, the Any part of the object element is decoded and used to access the appropriate data base entry. The program fragment of read_database that decodes the Any might be :

```
void read_database(read_result, read_argument)

ReadResult *read_result;
ReadArgument *read_argument;

{

    /* declare some control variables, an intermediate pointer, a common
        name variable and other possible types that might appear in the
        Attribute portion of "object" */

    int i, j;
    long attribute_id;
    RelativeDistinguishedName *rdn;
    PrintableString(64) common_name;
                            .
                            .
                            .

    /* get an intermediate pointer for cleaner looking code */
```

```
rdn = &read_argument->un.read_Argument.object.un.RDNSequence.val[0];

/* loop through the attributes, extracting the values in the
    attribute_value element */

for(i = 1; i < rdn->len; i++)
    {

    /* get the object identifier. Use the last component to decide the type of
        the attribute. Initialize the "common_name" counter */

    j = (rdn->val[i].attribute_type.size) - 1;

    attribute_id = rdn->val[i].attribute_type.comp[j];

    switch(attribute_id)
        {
        case 0 : /* Object Class */
                    .
                    .
                    .
                break;

        case 1 : /* Aliased Object Name */
                    .
                    .
                    .
                break;

        case 2 : /* Knowledge Information */
                    .
                    .
                    .
                break;
```

```
        case 3 : /* Common Name */

                decodeAny(X_500_DAP_interface, PrintableString(64),
                          common_name, rdn->val[i].attribute_value);
                break;

        case 4 : /* Surname */
                        .
                        .
                        .
        }

/* Access the data base and set read_result accordingly */
                        .
                        .
                        .
    } /* end of for loop */

/* free the space allocated by the runtime system for the Relative
    Distinguished Name */

    rofreei("ReadArgument", read_argument, X_500_DAP_interface);
}
```

VI. Miscellaneous Features

ELROS includes other functionality to control certain aspects of a distributed application. This chapter covers these features, which do not fit under any previous topic.

6.1 Fully specified type names

One problem that arises in applications written using ELROS that import multiple interfaces is type name collisions. They occur when two interfaces use the same type identifier. ELROS provides a way to disambiguate references to such types by prefixing the interface name to the type, delimiting it from the type name by a double colon.

For example, consider the specification for two interfaces – interfaceA and interfaceB, located respectively in files "interfaceA.i" and "interfaceB.i":

```
interface interfaceA;

Record
    {
    Real(14,7) x;
    Real(14,7)  y;
    } XYCoordinate;
          .
          .
          .
end;

interface interfaceB;

Record
    {
    Integer(4) x;
    Integer(4)  y;
```

```
    } XYCoordinate;
            .
            .
            .
end;
```

An embedded language program that imports both of these interfaces cannot declare variables of type XYCoordinate, since it is ambiguous to the ELROS preprocessor which type is being declared. To disambiguate these declarations, the programmer gives the appropriate interface name, followed by a double colon and then the type name.

For example, consider the following program :

```
#include  "interfaceA.i"
#include  "interfaceB.i"

XYCoordinate  xy_coordinate;
            .
            .
            .
```

The declaration of xy_coordinate will produce an ELROS preprocessor error message indicating that the declaration is ambiguous. However, suppose the following code is used :

```
#include  "interfaceA.i"
#include  "interfaceB.i"

interfaceA::XYCoordinate  xy_coordinate;
            .
            .
            .
```

Prefixing the typename by the interface name disambiguates the type identifier, allowing the ELROS preprocessor to properly parse the declaration.

6.2 Managing Remote Operations

There are two features in ELROS we have not discussed that are useful for managing remote operations. These are specifying application and presentation context names, and managing invoke and linked ids.

6.2.1 Application and presentation context names

In order for a computational unit to understand the data it receives, it must know the application protocol definitions its correspondent is using. Two names carried by the protocol's underlying remote operations identify these. The first is the presentation context name, which identifies the set of data structures being used by a correspondent. The second is the application context name, which identifies how these data structures will be used. Both of these names are ObjectIdentifiers.

ELROS provides a mechanism to specify the presentation and application context name associated with a particular interface. This is accomplished by including the values of these identifiers in an optional field of the **interface** definition statement. This field immediately precedes the optional field that specifies the autotagging default. In particular, the complete definition of the **interface** definition statement appears as follows (the square brackets indicate optional fields; they are not literals appearing in the interface identification statement) :

interface *interface-name* [(*application-context-name*,
presentation-context-name)]
[autotag | noautotag];

Both the *presentation-context-name* and *application-context-name* must be ObjectIdentifier constants formed according to the syntax given in Appendix C. When an instance that is tied to *interface-name* is bound to a correspondent (by executing **bind_addr**), these context identifiers are provided to the correspondent's runtime system to identify the services selected. The correspondent's runtime system compares them with the values provided when the correspondent performed its **bind_addr** and rejects the bind request if they are not equal. If the programmer does not provide a *presentation-context-name* and *application-context-name* in the interface specification, the ELROS preprocessor uses a default value.

6.2.2 Managing invoke and linked ids

The remote operations service supports the labeling of operations with values called operation identifiers (operation ids). These are integer values that identify individual operations, allowing an application receiving either a **result**, **error** or **reject** operation, each of which carries the identifier of the invoke operation that generated it, to match it to the appropriate service request. The operation ids on **invoke**, **result**, **error** and reject operations, are called invoke identifiers (invoke ids)[22]. In addition, a performer is allowed to return an **invoke** operation in response to one it receives from an invoker, in order to obtain further information needed to satisfy the request. The returning **invoke** operation carries not only its own invoke id, but also the invoke id of the original request, the latter value being known as the linked identifier (linked id).

ELROS supports two forms of invoke id management. In the first form, the application programmer relies on the ELROS runtime system to generate invoke ids. It does this by means of a per-instance internal state variable that it initializes to 1 when the communications instance is bound, uses as the source of the invoke id for each invoke operation, and increments by 1 after each use. Invoke ids on results, errors and rejects come from another per-instance internal variable that is set to the invoke id of the last invoke operation received.

Using this style of invoke id management, the application programmer can choose to ignore invoke ids, which is convenient when employing a synchronous style of remote operations, or when ensuring the correctness of the invoke id of a **result**, **error** or **reject** by comparing it with the number of operations sent since the instance was bound.

---

[22] We here follow the ISO terminology, which unfortuntately is somewhat confusing. The invoke id on an invoke operation refers to that operation. An invoke id on a result, error or reject operation refers to the operation id of the invoke operation that generated the error, result or reject.

The second form of invoke id management places the responsibility of choosing an invoke id value in the hands of the programmer. This management style uses optional fields in the **give** and **accept** statement to specify and obtain invoke id values.

The programmer specifies an invoke id value for a particular operation by including an integer expression in square brackets immediately following the name of an operation reference. For example, to specify an invoke id on a ReadOp reference within a **give** statement the following code could be used :

```
give  X_500_performer
        {
        ReadOp[invoke_id]  (read_request);
        }
```

In this fragment, the value in the integer variable "invoke_id" is used to set the invoke_id field of the ReadOp operation sent to the performer. The performer acts on the ReadOp request and returns a result to the invoker using the same operation id that labeled the ReadOp request. The program fragment to accomplish might look like :

```
give  X_500_invoker
        {
        ReadResultOp[invoke_id]  (read_result);
        }
```

Here the variable invoke_id should contain the operation id value the performer received in the corresponding ReadOp request.

When a correspondent receives an operation request within an **accept** statement, a pointer to its invoke id is stored in a variable supplied by the programmer. This pointer variable is placed in square brackets after the reserved word **accept**. For example, an X.500 performer would normally execute the following **accept** statement to obtain an operation request from an invoker :

```
long  *invoke_id;
```

```
accept [invoke_id] X_500_invoker
    {
    on ReadOp(read_argument)
          {
            /* call the subroutine that performs a read on the X_500
                database */

          read_database(&read_result, &read_argument);

          /* return the read result to the invoker */

          give X_500_invoker
                {
                ReadResultOp [*invoke_id] (read_result);
                }
          }
    }
```

Notice that a pointer to the invoke id of the ReadOp operation is stored in the variable "invoke_id", this value being used by the ReadResultOp operation when the result is returned.

An operation reference within the context of a **give** statement may also take a pointer variable to receive a pointer to a linked id. This variable is placed within the square brackets after the operation identifier, separated from the invoke id pointer variable by a comma. If no linked id is received for the operation, the linked id pointer variable is set to null (i.e., zero).

The X.500 directory service does not use linked ids, so we invent a simple example to illustrate their use. Suppose a performer is written to make a real-time determination that each operation it receives comes from a legitimate invoker and is not the replay of a previous operation request from an active wiretapper. It does this by challenging each operation request with a random number returned to the invoker, under the expectation that the invoker will encrypt this number with a key it holds in common with the performer and return it to the performer.

When the performer receives an operation, it sends a challenge operation containing a random number to the invoker with its linked id set to the invoke id of the original operation. It expects to receive a response to the challenge containing the encrypted value of the random number.

The performer program fragment might look like the following :

```
long *invoke_id, *linked_id, challenge_id;

accept [invoke_id] invoker
    {

    /* wait for an Op request */

    on Op(op_argument)
          {

          /* challenge the invoker and wait for its response */

          random_number = rand();

          give invoker
                {

                /* send the challenge */

                ChallengeOp [++challenge_id, *invoke_id] (random_number);

                /* get back the response from the invoker */

                accept [linked_id] invoker
                    {
                    on ResponseOp(response)
                        {

                        /* if the invoke_id of the original Op request equals
                            the invoke_id on the response and the response
```

```
          argument equals the encrypted value, do the Op
          request */

        if(*linked_id == challenge_id
                    &&
        response == encrypt(random_number))
            {

            /* perform semantics of Op */

            }
        else
            {

            /* Otherwise, inform the invoker that the Op
                request failed because the response was bad */

            give invoker
                {
                ErrorOp [challenge_id] ()
                }
            }
        } /* end of inner on-clause */
      } /* end of inner accept statement */
    } /* end of the give statement */
  } /* end of the outer on-clause */
} /* end of the outer accept statement */
```

This program fragment assumes that Op and ChallengeOp are invokes, ResponseOp is a result and ErrorOp is an error. It also assumes that challenge_id is initialized outside the fragment. The instance variable "invoker" is assumed bound by a previous execution of the **bind_addr** statement.

The invoker must be able to receive the ChallengeOp sent by the performer and check the linked id it contains to ensure it equals the invoke id that it sent previously. The invoker gets the linked id by providing a second pointer variable in square brackets after the reserved word **accept**. This variable is separated by a comma from the variable that receives the invoke id pointer. Using the example described above, the invoker program fragment might look like :

```
long invoke_id, *linked_id, *challenge_id;

give performer
    {

    /* request Op service from performer */

    Op [invoke_id] (op_argument);

    /* wait for the challenge to arrive. Use a loop so that spurious
        challenges are rejected */

    looping = YES;

    while(looping)
        {
        accept [challenge_id, linked_id] performer
            {
            on ChallengeOp(random_number)
                {

                /* make sure this challenge is for the right invocation */

                if(*linked_id == invoke_id)
                    {

                    /* if so, compute the response and return it to the
                        performer */

                    response = encrypt(random_number);
```

```
                give performer
                    {
                    ResponseOp [*challenge_id] (response);
                    }

                /* break out of the loop */

                looping = NO;

                }
            else
                {
                /* ignore the challenge and loop so the correct challenge
                    is accepted */

                continue;
                }
            } /* end of on-clause */
        } /* end of accept statement */
    } /* end of while loop */
} /* end of give statement */
```

Once again, this fragment assumes the initialization of a number of variables as well as the binding of the instance performer outside the program fragment.

6.3 Communication Instance Support Routines and Statements

ELROS includes a set of routines and statements that allow the programmer to finely control activity on a communications instance. We document these routines in this section.

6.3.1 set_timeout()

When a communications instance is bound to a correspondent using **bind_addr**, a timeout interval is given that applies to all blocking activity on the instance. Examples of blocking activity are **accept** statements and calls by the ELROS runtime system to internal blocking communications primitives. Sometimes the programmer desires to change this value to either increase or decrease the amount of time an activity will block before causing a timeout error (represented as one of the **com_error** exceptions). The routine set_timeout() is provided for this purpose. Its interface definition is :

void set_timeout(*instance_variable*, *timeout-value*)

The *instance_variable* argument is an active instance variable and *timeout-value* is the number of seconds that the ELROS runtime system should wait before taking a timeout **com_error** exception.

Set_timeout() is also useful for disabling and enabling timeouts on all instances. If *instance_variable* equals NULL (i.e., zero) and *timeout-value* is zero, timeouts on all instances are disabled. If *instance_variable* equals NULL (i.e., zero) and *timeout-value* is non-zero, timeouts on all instances are enabled, using the value previously established for each instance, rather than the value given in the enabling set_timeout() call. This use of set_timeout() is especially useful when the programmer wants to disable and then re-enable timeouts for debugging purposes. In such situations, set_timeout() can be called from a debugger with a NULL *instance_variable*.

6.3.2 **abort_inst()**

Normally, a programmer will call **unbind** to close communications with a correspondent. There are occasions, however, when **unbind** does not provide the correct semantics.

For example, if a responder wishes to close an instance, the underlying communications semantics of **unbind** may not allow this. In accordance with the remote operations standard, an **unbind** executed by an initiator causes an A-Associate.request to be transmitted to the responder. However, an **unbind** executed by a responder does not cause such a transmission. Instead, the responder blocks waiting for the initiator to unbind. The **abort_inst** statement causes an A-Associate.abort request to be sent irregardless of what type of instance is specified and whether the application executing the statement is the initiator or responder.

The format of an abort_inst call is :

abort_inst(*instance_variable*);

The single argument of **abort_inst**, *instance_variable*, is an active communications instance variable.

6.3.3 **flush_inst()**

There are situations in which data is in an internal buffer associated with an instance waiting to be transmitted and the correct operation of the application requires it to be sent immediately. For example, if an **accept** statement using one instance is embedded within a **give** statement using a different instance, any operations referenced before the **accept** statement is executed will not be sent to the correspondent (since ELROS batches operations and sends them only when control passes out of the **give** statement)[23]. If the **accept** statement depends on activity initiated due to operations sent within the **give** statement, the **accept** statement will timeout, since the responder will not receive the operations and consequently not return operations over the **accept** instance. The programmer must force ELROS to send the operations before the **accept** statement is executed. This is achieved by executing the **flush_inst** statement :

flush_inst(*instance_variable*);

The argument to **flush_inst**, *instance_variable*, is an active communications instance variable.

Both **abort_inst** and **flush_inst** are ELROS statements, rather than ELROS standard subroutines. Since **abort_inst** never fails, there is no need to catch exceptions it might cause. However, **flush_inst** can fail and so takes an

---

[23] If the **give** and **accept** statements use the same instance, the code generated by the preprocessor calls flush_inst before control passes into the **accept** statement.

optional **exceptions** clause. Exceptions raised by **flush_inst** include **com_error**, **internal_error** and **protocol_error**.

6.3.4 get_invokeID()

ELROS supports two forms of operation id management. If the application programmer selects not to specify an operation id for operation references inside **give** statements, the ELROS runtime system uses a per-instance internal state variable to generate these identifiers. This style of programming is useful when the application protocol is synchronous, i.e., the invoker waits for a reply to one request before sending another, and duplicate operations cause no harm, i.e., operations are idempotent.

The other form of operation id management requires the user to manage the operation id sequence, specify the appropriate operation id in each operation reference that occurs within a **give** statement, and check the operation id on each operation received in an **accept** statement. Specifying an operation id in an operation reference signals the ELROS runtime system that it should not generate the operation id from the per-instance internal state.

Sometimes the programmer wishes to change management techniques in the middle of an operation sequence on an active instance. To move from an ELROS managed operation id sequence to an application managed sequence requires the program to ascertain the internal per-instance value for the next operation id. The routine get_invokeID() does this. Its interface definition is :

long get_invokeID(*instance_variable*);

get_invokeID() takes an active *instance_variable* as its argument and returns an invokeID, a long integer, as its result.

6.3.5 set_invokeID()

If the programmer wishes to change the operation id management technique on an active instance from application managed to ELROS runtime system managed, there needs to be a way to set the internal per-instance value that will be used by the runtime system as the next operation id. Sometimes this is relatively straight forward, since the ELROS runtime system loads this internal value with the value supplied by the programmer in an operation reference. If the programmer elects to discontinue specifying an operation id in operation references, the runtime system uses the last value specified to begin its managed sequence.

In other situations the application programmer wants to reset the internal value without sending an operation. The routine set_invokeID() does this. Its interface definition is :

void set_invokeID(*instance_variable*, *invokeID*);

The first argument to set_invokeID is an active instance variable and the second argument is an invokeID, which is of type long.

6.3.6 **get_addr()**

ELROS is structured so that instance values, i.e., the value inside instance variables, can be passed into and returned from C procedures[24]. Sometimes it is desirable for a subroutine to perform the **bind_addr** statement on an instance variable. This hides the address information from the caller of the subroutine.

When programmers use this type of program modularity, some modules of the application may not have access to the addressing information that bound an instance variable. If the application needs to open another instance to the same correspondent, a way of recovering the address information associated with a bound instance is required.

The **get_addr** statement is provided for this purpose. Its syntax is :

---

[24] Currently, this feature is not fully supported in ELROS. **give**, **accept**, **encode** and **decode** statements require instance variable names, not expressions that evaluate to instance values. To use an instance value passed between two procedures in either of these statements, it must be assigned to an instance variable declared within the using procedure and tied to the correct interface. No checking is performend by the ELROS runtime system to ensure an instance value is of the correct interface type when it is assigned to an instance variable. It is up to the programmer to keep things straight.

get_addr(*instance_variable*, *addr_struct_pointer*)

The first argument to **get_addr** is an active instance and the second argument is a pointer to a bind_address that receives the addressing information associated with the instance.

6.3.7 **bind_mem**

The ELROS statement **bind_addr** is used to establish communications between two correspondents. Sometimes, for debugging or other purposes, a programmer wishes to capture the byte stream that the ELROS runtime system produces in response to the execution of an ELROS program. The ELROS statement **bind_mem** supports this requirement. The syntax of **bind_mem** is :

bind_mem(*instance_variable*, *rules*, *Any_variable*, *openstring*)

Execution of the **bind_mem** statement binds *instance_variable* to the area of memory described by the *Any_variable* argument, which is a C struct with the following format :

```
typedef struct any
        {
        unsigned char *pointer;
        unsigned long count;
        unsigned long size;
        struct any *sig;
        } Any;
```

As with the **bind_addr** statement, the *rules* argument takes only one value at present, i.e., asn_1_ber. The *openstring* argument is a single letter string, either "r" or "w". The "w" value specifies that the runtime system output should be written to memory. The "r" value specifies that a memory buffer written previously by the runtime system should be read[25].

If the "w" value of *openstring* is specified, the programmer passes a pointer to an uninitialized **Any** variable to **bind_mem,** which allocates storage from the heap and initializes the elements of the variable to the appropriate values. The "pointer" element is the address of the memory buffer allocated, "count" is the number of bytes containing ELROS runtime system output, and "size" is the number of bytes in the allocated buffer. The component “sig” is used for internal checking. If the size of the buffer is exceeded during output, the runtime system allocates a new buffer twice as big as the current one, copies the data from the old buffer to the new buffer, frees the old buffer and writes the remaining output to the new buffer. To ensure correct operation, the programmer should not access the elements of the **Any** variable until the communications instance is unbound with an **unbind** statement.

If the "r" value of *openstring* is specified, the programmer passes an Any variable to **bind_mem.** This variable should reference a memory buffer previously written by the ELROS runtime system. When the communications instance generates a **done** exception, the programmer should free the buffer, using the **clearAny** macro.

Since **bind_mem** is a normal ELROS statement, it can generate exceptions. The programmer catches these exceptions with an **exceptions** clause, which takes the same on-clauses as **bind_addr**. **bind_mem** is used to implement the "Any" macros discussed in chapter 5.

6.3.8 **bind_file**

ELROS also supports input and output of runtime system data from/to a file. To bind a communications instance to a file, the programmer uses the **bind_file** statement, which has the following syntax :

bind_file(*instance_variable*, *rules*, *infilename*, *outfilename*)

---

[25] Currently, **bind_mem** is limited to establishing an instance that either reads from memory or writes to memory. Thus, **bind_mem** establishes a simplex channel, rather than the duplex channel established between correspondents by **bind_addr**.

The communications instance represented by *instance_variable* is bound as follows. Output generated by the ELROS runtime system for the instance is written to *outfilename*, a quoted pathname. Input for the instance is taken by the runtime system from *infilename*, a quoted pathname. The *rules* argument takes only one value at present, asn_1_ber.

The **bind_file** statement takes an optional **exceptions** clause, which may contain any of the on-clauses used for **bind_addr** and **bind_mem**.

VII. Debugging

Writing a distributed application is generally the easiest part of its development. Getting a distributed application to work correctly is on average much more time consuming.

ELROS provides several tools that assist the applications programmer in debugging his code. These fall into three classes: 1) routines generated by the preprocessor to print variables declared as ELROS types, 2) a set of logging routines, and 3) a set of utilities for viewing and filtering the log data generated by these routines. The print and logging routines can be called from within an ELROS program or from a standard debugger such as dbx.

7.1 ELROS generated print routines

The ELROS preprocessor generates print routines for the basic ELROS types as well as for each type defined in an interface. These routines are accessed through one of four standard routines: 1) roprinti(), 2) roprint(), 3) froprinti(), and 4) froprint().

Roprinti() is called with the following parameters :

```
roprinti("type", &variable, interface)
```

The first argument is the ELROS type of the variable. The type is placed within double quotes so that the routine can be called from a debugger like dbx, which doesn't have ELROS types available in its symbol table. The second argument is the address of the variable to print. The third argument is the interface from which the type is drawn.

For example suppose we wish to print the value of a variable entry_modification of type EntryModification from the X_500 interface. From an embedded language program the call would look like :

```
roprinti("EntryModification", &entry_modification,
                                        X_500_DAP_interface);
```

Note that the interface identifier is **not** surrounded by double quote marks. This is possible because the ELROS preprocessor declares a variable with the same name as the interface, which is included in its generated code and thus appears in the symbol table of the debugger.

To print the value of entry_modification from the dbx debugger the following dbx command is executed (the prompt dbx> is produced by dbx and not typed) :

```
dbx> call roprinti("EntryModification" &entry_modification,
                                    X_500_DAP_interface);
```

The result of this call is (assuming the Union indicates the removeAttribute element and its value is the ObjectIdentifier of "common name") :

```
= {
    [removeAttribute] =
        2.5.4.3
    }
```

The string "removeAttribute" in square brackets at the beginning of the output text indicates which of the Union components is present in the variable entry_modification. The string "2.5.4.3" is the value of this component.

The ELROS runtime system must know the interface from which a type is drawn in order to call the correct routine to print its values (e.g., two interfaces imported into the embedded language code might define types of the same name). roprinti() specifies the interface name through its third argument. The ELROS runtime system remembers this interface name in an internal variable, allowing a subsequent call of roprint() :

```
roprint("type", &variable)
```

The first two arguments of roprint() are the same as roprinti(), the difference between the two routines being that the former does not specify an interface. Instead it uses the interface stored in the ELROS runtime system internal variable set by roprinti() (or froprinti() – see below). The use of roprint() relieves the programmer from typing the same interface name over and over again.

The output from both roprint() and roprinti() is directed to stdout. Sometimes it is desirable to print the value of an ELROS variable to a file. The routines froprinti() and froprint() accomplish this. Froprinti() takes four arguments. The first argument is a file pointer to an open file into which the value of the variable is written. The last three arguments are the same as the three arguments of roprinti(). For example, the following code prints the value of the variable entry_modification to the file "~nessett/dump" :

```
FILE *fd;

fd = fopen("~nessett/dump","w");

froprinti(fd, "EntryModification",  &entry_modification,
                                    X_500_DAP_interface);
                        .
                        .
                        .
```

The routine froprint() operates like roprint() except the value of the variable is printed to the file described by its first argument, rather than to stdout. Suppose there are two variables of type EntryModification, entry_modification1 and entry_modification2. The following code prints the values of these variables to the file "~nessett/dump":

```
FILE *fd;

fd = fopen("~nessett/dump","w");

froprinti(fd, "EntryModification",  &entry_modification1,
                                    X_500_DAP_interface);
```

```
froprint(fd, "EntryModification", &entry_modification2);
                    .
                    .
                    .
```

7.2 Tracing ELROS program execution

In addition to routines to print ELROS data types, the ELROS package includes routines to trace the execution of ELROS embedded language statements. There are two sets of tracing routines : 1) the set used for tracing remote operation execution, and 2) the set used for tracing the generation and consumption of ASN.1. The first set is called level 7 tracing, while the second set is called level 6 tracing.

7.2.1 Level 7 tracing

Four routines are provided to trace remote operations activity : 1) start_log7i(), start_log7(), 3) stop_log7i(), and 4) stop_log7(). When a programmer desires to generate a trace of the remote operations sent and received by a program, he calls start_log7i(). Normally start_log7i() is called from a debugger, although it may also be called from an embedded language program. The routine takes two parameters : 1) an active instance (i.e., one which has appeared in a bind statement) and 2) the name of a file to which trace information is output. The first parameter allows the programmer to specify a specific instance on which to conduct tracing. Tracing on that instance remains in effect until the instance is unbound or until stop_log7i() is called. If the first parameter to start_log7i() is NULL (i.e., zero), global tracing at level 7 is enabled. In that case level 7 activity on all instances is traced.

The second parameter to start_log7i() is the name of the file, enclosed in double quote marks, that will receive the tracing information. If start_log7i() is called with a NULL second argument, the following set of rules determines the trace output file. First, if start_log7i() has been called previously, the ELROS runtime system uses the file established by that call. Trace information is appended to it. Otherwise, an environmental variable, RO_LOG7FILE is examined. If it contains a name, trace output is sent there. If RO_LOG7FILE is null, a default name is used – ELROS.log7. Note these rules imply that if the file used by the ELROS runtime system already exists when start_log7i() is called, then trace information will be appended to whatever is in the file.

The routine start_log7(), taking only a filename enclosed in double quotes, turns on level 7 global logging. It uses the same filename conventions as start_log7i().

When a programmer wishes to turn off tracing, he calls stop_log7i(). Stop_log7i() takes a single parameter, an instance. Tracing for that instance is deactivated, while tracing for all other instances remains active. If the instance argument is NULL, level 7 global tracing is deactivated. The routine stop_log7() takes no parameter and turns off level 7 global tracing (although tracing on specific instances activated by calling start_log7i() remains in effect).

To illustrate the use of these routines, suppose a programmer is debugging an ELROS embedded language program from dbx. The following command starts tracing on the instance "inst" with tracing information going to the file "~nessett/trace".

```
dbx> call start_log7i(inst, "~nessett/trace")
```

The resulting trace information would appear as follows :

```
A00004ab4:    302  X_500_invoker.c  X_500_performer
A00004ab4:         ENTERING give statement
A00004ab4:    305  X_500_invoker.c  X_500_performer
A00004ab4:         GIVING ReadOp(invoke_ID=12, linked_ID=0)
A00004ab4:              read_argument = {
A00004ab4:                   [ReadArgumentType] =
A00004ab4:                        object = {
                          .
                          .
                          .
A00004ab4:    312  X_500_invoker.c  X_500_performer
                   EXITING give statement
```

The first field in every line is an integer constant that identifies the particular instance on which the activity occurred. Each trace entry consists of multiple lines, the first of which indicates the line number in the source file where the trace entry was generated, followed by the source file name and the name of the instance variable used in the traced activity. Subsequent lines describe the activity in detail.

7.2.2 Level 6 tracing

Four routines are provided to trace ASN.1 encoding and decoding activity : 1) start_log6i(), start_log6(), 3) stop_log6i(), and 4) stop_log6(). When a programmer desires to generate a trace of the ASN.1 sent and received by a program, he calls start_log6i(). Normally start_log6i() is called from a debugger, although it may also be called from an embedded language program. The routine takes two parameters : 1) an active instance and 2) the name of a file to which trace information is output. The first parameter allows the programmer to specify a specific instance on which to conduct tracing. Tracing on that instance remains in effect until the instance is unbound or until stop_log6i() is called. If the first parameter to start_log6i() is NULL (i.e., zero), level 6 global tracing is enabled. In that case activity on all instances is traced.

The second parameter to start_log6i() is the name of the file, enclosed in double quote marks, that will receive the tracing information. If start_log6i() is called with a NULL second argument, the following set of rules determines the trace output file. First, if start_log6i() has been called previously, the ELROS runtime system uses the file established by that call. Trace information is appended to it. Otherwise, an environmental variable, RO_LOG6FILE is examined. If it contains a name, trace output is sent there. If RO_LOG6FILE is null, a default name is used – ELROS.log6. Note these rules imply that if the file used by the ELROS runtime system already exists when start_log7i() is called, then trace information will be appended to whatever is in the file.

The routine start_log6(), taking only a filename enclosed in double quotes, turns on level 6 global logging. It uses the same filename conventions as start_log6i().

When a programmer wishes to turn off tracing, he calls stop_log6i(). Stop_log6i() takes a single parameter, an instance. Tracing for that instance is deactivated, while tracing for all other instances remains active. If the instance argument is NULL, level 6 global tracing is deactivated. The routine stop_log6() takes no parameter and turns off level 6 global tracing (although tracing on specific instances activated by calling start_log6i() remains in effect).

To illustrate the use of these routines, suppose a programmer is debugging an ELROS embedded language program from dbx. The following command starts tracing on the instance "inst" with tracing information going to the file "~nessett/trace".

```
dbx>  call start_log6i(inst, "~nessett/trace")
```

The resulting trace information would appear as follows :

```
N00091254:
C00091254:BIND: platform=ISODE, role=RESPONDER
C00091254:BIND: calling_address=128.115.80.31+18002/#0
C00091254:BIND: called_address=128.115.80.31+18002/#0

R00091254: 61 25 30 23 02 01 01 a0 1e a1 1c 02 01 01 02 01
R00091254: 21 30 14 30 08 13 02 32 31 13 02 34 35 30 08 13
R00091254: 02 38 37 13 02 39 39

S00091254: 61 80 30 80 02 01 01 a0 80 a2 80 02 01 01 30 80
S00091254: 02 01 21 31 80 a1 80 30 80 30 80 02 01 15 02 01
S00091254: 2d 00 00 30 80 02 01 57 02 01 63 00 00 00 00 00
S00091254: 00 a2 80 30 80 30 80 02 01 15 02 01 2d 00 00 30
S00091254: 80 02 01 57 02 01 63 00 00 00 00 00 00 a3 80 30
                              .
                              .
                              .
S00091254: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S00091254: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S00091254: 00 00 00 00 00 00 00 00 00
```

C00091254:UNBIND

The first field in every line is an integer constant that identifies the particular instance on which the activity occurred. The letter that precedes the integer specifies what kind of information is contained on the line. The meaning of these letters are : 1) N – the creation of an instance, 2) R – information received on the instance, 3) S – information sent over the instance, and 4) C – a comment generated by the runtime system (in the above example these comments identify the binding and unbinding of the instance).

7.3 Using the ELROS tracing routines in realtime

Sometimes a programmer debugging an ELROS embedded language program desires to see the execution of ELROS statements in realtime as they occur. Two utilities are provided to support this style of debugging : 1) logfilter and 2) loganal.

Logfilter is a utility that accepts an ELROS trace file and filters out unwanted information. When filtering level 7 traces, its execute line is :

logfilter -a *filename @instance*

Filename is the name of a file containing ELROS runtime system trace information. If *filename* isn't specified, stdin is used. *Instance* is an integer constant that identifies an instance in a trace file (i.e., it is the integer constant that appears after the "A" on each line of the trace information). The output of logfilter is sent to stdout, so the programmer can view it on his terminal. If *instance* is not specified (i.e., the *@instance* is left off the execute line), all information in the trace file is sent to stdout with the instance information (i.e., the A00004ab4 which prefixes each line in the above example) stripped off.

When filtering level 6 traces, the appropriate execute line is :

logfilter -r *filename @instance*

to analyze the data received on *instance*, or logfilter -s *filename* @*instance*

to analyze the data sent on *instance*.
The *filename* and *instance* arguments act in the same way as for level 7 filtering, except the instance information passed through the filter are those lines starting with C, N, R and S.

Logfilter is useful when a complete file of trace information is available for analysis. To view trace information as it is generated requires the use of the second utility – loganal.

To use loganal to analyze level 7 traces, the programmer opens a second window and types :

loganal *filename* | logfilter -a @*instance* | more

Use of this execute line is predicated on the assumption that *filename* is the file established by the programmer when he called start_log7i() or start_log7(). Loganal opens up a pipe to *filename* and reads its contents as they are written. Each line read is sent to stdout, which in the execute line given above pipes it to logfilter. Logfilter discards all information unrelated to the instance *instance* and pipes the result to more.

To examine all level 7 tracing information generated by the ELROS runtime system, the programmer should use the following execute line :

loganal *filename* | logfilter -a| more

To use loganal to analyze level 6 traces, the programmer opens a second window and types :

loganal *filename* | logfilter -r @*instance* | dasn | more

to analyze the data received on *instance*, or

loganal *filename* | logfilter -s @*instance* | dasn | more

to analyze the data sent on *instance*.

The utility "dasn" is a context free asn.1 translator that converts the hexadecimal representation in a level 6 trace file to an ascii representation that is easier to read and understand.

To examine all level 6 tracing information generated by the ELROS runtime system, the programmer should use one of the following execute lines :

loganal *filename* | logfilter -r| dasn | more

to view the data received on all instances, or

loganal *filename* | logfilter -s| dasn | more

to view the data sent on all instances.

VIII. ELROS and Multithreading

ELROS is designed and implemented so it can be used in multithreaded programs. Currently, it is integrated into only one threads implementation, the PADS tasking package[26] developed by the Livermore Computer Center at Lawrence Livermore National Laboratory[27]. This chapter discusses several aspects of ELROS programming that are related to its use in a PADS tasking environment. It is the intent of the ELROS project to integrate ELROS with POSIX 1003.4a Pthreads implementations, when they become widely available. It is also possible to MAKE a version of ELROS that does not use PADS.

8.1 PADS

The PADS tasking interface is simple and fairly easy to use. There is one basic type exported by the PADS tasking package – Task. A task is created by calling the procedure Newtask which has the following interface :

Task Newtask(*procedure*, *variable*)

The first argument to Newtask() is the address of a *procedure* to execute when the task is given control for the first time. The second argument is a *variable* address that is passed to this procedure. These two arguments are declared as follows :

void (**procedure*)();
unsigned char **variable*;

The procedure Newtask() returns a Task value, which is the identifier of the task created.

---

[26] The terms "thread" and "task" have no wide-spread common definitions. In this tutorial they are taken to be synonymous.

[27] The PADS library supports a communications service as well as a tasking package. ELROS does not make use of the communications portion of PADS. Only the tasking package is used.

A task finds out its own identifier by calling the procedure Ego(). Ego() takes no arguments and returns a value of type Task. The procedure Sleep() suspends execution of the task that calls it, allowing other tasks to run. It also takes no argument, and returns the current time (the task is suspended in Sleep() until it wakes up). The procedure Wake() takes a Task value as its argument, placing the task identified on the ready-to-run queue. The procedure Alarm() takes an unsigned long as its argument. It marks a future point in time, equal to the time specified by its argument at which the task should be awakened. If the time given by the argument is less than the current time, Alarm() is effectively a no-op. Normally, a task calls Alarm() and then immediately calls Sleep().

Each task has a variable associated with it that is used to point to private data. This variable is unsigned char *Tvarb. Finally, an external variable, unsigned long Stksize, is available for the programmer to control the number of bytes allocated for the stack of a newly created task.

In summary, the PADS interface is :

Task Newtask(*procedure*, *variable*)
    void (**procedure*)();
    unsigned char **variable*;

Task Ego();

unsigned long Sleep();

void Wake(task)
    Task task;

void Alarm(time)
    unsigned long time;

unsigned char *Tvarb;

unsigned long Stksize;

8.2 ELROS I/O and PADS

The PADS tasking package includes a scheduler that determines the next task to run after another task gives up control. In order to allow the use of UNIX I/O by the procedures executing within a task, all tasks must use non-blocking I/O, relying on the scheduler to perform a select() when no task remains in the ready-to-run queue. The scheduler gains control after select() returns and places those tasks with I/O completions on the ready-to-run queue.

ELROS is integrated into the select() strategy of the PADS task scheduler. However, it is sometimes desirable for a task to perform I/O for some other purpose than sending and receiving messages. To harmonize the use of non-ELROS I/O with ELROS-related I/O (e.g., the I/O that occurs as the result of a **give** or **accept** statement execution), a task must register the sockets it uses for non-ELROS I/O with the ELROS runtime system. This is achieved by calling the procedure Add_descriptor, which is defined as follows :

Add_descriptor(*file-descriptor*, *event-direction*)

file-descriptor is added to the ELROS runtime system select mask by this call. The argument *event-direction* takes one of three values : 1) _ _ ROm_WRITEDIR, 2) _ _ ROm_READDIR, and 3) _ _ ROm_ERRORDIR. This argument informs the ELROS runtime system the direction of the event it must wait for (and thus identifies which select mask to use). Once the user task makes a non-blocking I/O call and calls Add_descriptor, it normally would call Alarm(), to set a timeout wakeup call, and then Sleep(). When the procedure Sleep() returns control to the task, it is the task's responsibility to check the status of the socket to determine if errors occurred during the select() performed by the ELROS runtime system.

Appendix A - An ASN.1 Fragment From the X.500 Recommendations

The following ASN.1 represents all the definitions required by the Read operation of the Directory Access Protocol (DAP). To minimize the space required to display these definitions and to improve clarity, many definitions of types used only once in the fragment have been eliminated and expanded in-line. Constant definitions have been eliminated from INTEGER and ENUMERATED types when these definitions are not used elsewhere in the fragment. The ASN.1 Signed and Signature macros have been expanded in-line. Finally, the definitions of ReadArgument and ReadResult use intermediate types, ReadArgumentType and ReadResultType, not found in the standard in order to allow the Signed macro that encapsulates ReadArgument to be expanded in a reasonable amount of space.

```
/* From X.208 */

    UTCTime                     ::=    [UNIVERSAL 23] IMPLICIT VisibleString

/* From X.501 */

    AttributeValue              ::=    ANY

    AttributeType               ::=    OBJECT IDENTIFER

    Attribute                   ::=    SEQUENCE {
                                       type          AttributeType
                                       values        SET OF AttributeValue
                                       }

    AttributeValueAssertion     ::=    SEQUENCE {AttributeType,
                                                  AttributeValue}

    RelativeDistinguishedName ::=      SET OF AttributeValueAssertion
```

```
RDNSequence            ::=  SEQUENCE OF
                                  RelativeDistinguishedName

Name                   ::=  CHOICE { RDNSequence }

DistinguishedName      ::=  RDNSequence

/* From X.509 */

AlgorithmIdentifier    ::=  SEQUENCE {
                                  algorithm OBJECT IDENTIFIER,
                                  parameters ANY OPTIONAL }

Validity               ::=  SEQUENCE { notBefore    UTCTime,
                                       notAfter     UTCTime }

SubjectPublicKeyInfo   ::=  SEQUENCE { algorithm AlgorithmIdentifier,
                                       subjectPublicKey BIT STRING }

Certificate            ::=  SEQUENCE {
                              SEQUENCE {
                                  version [0] INTEGER DEFAULT v1988,
                                  serialNumber INTEGER,
                                  signature AlgorithmIdentifier,
                                  issuer Name,
                                  validity Validity,
                                  subject Name,
                                  subjectPublicKeyInfo
                                           SubjectPublicKeyInfo },
                               AlgorithmIdentifier,
                              BIT STRING }

CertificatePair        ::=  SEQUENCE {
                                  forward [0] Certificate OPTIONAL,
                                  reverse [1] Certificate OPTIONAL }
```

```
CertificationPath          ::=    SEQUENCE {
                                      userCertificate Certificate,
                                      theCACertificates
                                          SEQUENCE OF CertificatePair
                                              OPTIONAL }

/* From X.520 */

PresentationAddress        ::=    SEQUENCE {
                                      pSelector [0] OCTET STRING
                                          OPTIONAL,
                                      sSelector [1] OCTET STRING
                                          OPTIONAL,
                                      tSelector [2] OCTET STRING
                                          OPTIONAL,
                                      nAddresses [3] SET SIZE(1...MAX)
                                              OF OCTET STRING }

/* From X.518 */

OperationProgress          ::=    SET {
                                      nameResolutionPhase [0]
                                          ENUMERATED {
                                              notStarted (1),
                                              proceeding (2),
                                              completed (3) },
                                      nextRDNToBeResolved [1]
                                          INTEGER OPTIONAL }

ReferenceType              ::=    ENUMERATED {
                                      superior (1),
                                      subordinate (2),
                                      cross (3),
                                      nonSpecificSubordinate (4) }
```

```
    AccessPoint                 ::=     SET {
                                            ae-title [0] Name,
                                            address [1] PresentationAddress }

    ContinuationReference       ::=     SET {
                                            targetObject [0] Name,
                                            aliasedRDNs [1] INTEGER
                                                OPTIONAL,
                                            operationProgress [2]
                                                OperationProgress,
                                            rdnsResolved [3] INTEGER
                                                OPTIONAL,
                                            referenceType [4] ReferenceType
                                                OPTIONAL,
                                            accessPoints [5] SET OF AccessPoint }

/* From X.511 */

    EntryInformationSelection   ::=     SET {
                                            attributeTypes CHOICE {
                                              allAttributes [0] NULL,
                                              select [1] SET OF AttributeType }
                                                DEFAULT allAttributes NULL,
                                            InfoTypes [2] INTEGER
                                                {attributeTypesOnly (0),
                                                 attributeTypesAndValues (1)}
                                                 DEFAULT
                                                    attributeTypesAndValues }

    EntryInformation            ::=     SEQUENCE {
                                            DistinguishedName,
                                            fromEntry BOOLEAN DEFAULT TRUE,
                                            SET OF
                                                {
                                                CHOICE {
                                                    AttributeType,
```

```
                                                Attribute}
                                                OPTIONAL } }

ServiceControls            ::=    SET {
                                        options [0] BIT STRING DEFAULT {},
                                        priority [1] INTEGER DEFAULT 1,
                                        timeLimit [2] INTEGER OPTIONAL,
                                        sizeLimit [3] INTEGER OPTIONAL,
                                        scopeOfReferral [4] INTEGER
                                            OPTIONAL }
ProtectionRequest          ::=    INTEGER

SecurityParameters         ::=    SET {
                                        certification-path [0] CertificationPath
                                            OPTIONAL,
                                        name [1] DistinguishedName
                                            OPTIONAL,
                                        time [2] UTCTime OPTIONAL,
                                        random [3] BIT STRING OPTIONAL,
                                        target [4] ProtectionRequest,
                                            OPTIONAL }

Extension                  ::=    SET {
                                        identifier [0] INTEGER,
                                        critical [1] BOOLEAN
                                            DEFAULT FALSE,
                                        item [2] ANY }

ReadArgumentType           ::=    SET {
                                        object [0] Name,
                                        selection [1] EntryInformationSelection
                                            DEFAULT {},
                                        [30] ServiceControls DEFAULT {},
                                        [29] SecurityParameters DEFAULT {},
                                        requestor [28]
                                            DistinguishedName OPTIONAL,
```

```
                                        [27] OperationProgress
                                            DEFAULT {notStarted}
                                        aliasedRDNs [26]
                                            INTEGER OPTIONAL,
                                        extensions [25] SET OF Extension
                                            OPTIONAL }

ReadArgument                ::=     CHOICE {
                                        ReadArgumentType,
                                        SEQUENCE {
                                            ReadArgumentType,
                                            AlgorithmIdentifier,
                                            BIT STRING }  }

ReadResultType              ::=     SET {
                                        entry [0] EntryInformation,
                                        [30] SecurityParameters DEFAULT {},
                                        performer [29]
                                            DistinguishedName OPTIONAL,
                                        [28] aliasDereferenced BOOLEAN
                                            DEFAULT TRUE }

ReadResult                  ::=     CHOICE {
                                        ReadResultType,
                                        SEQUENCE {
                                            ReadResultType,
                                            AlgorithmIdentifier,
                                            BIT STRING }  }

AttributeErrorParameter     ::=     SET {
                                        object [0] Name,
                                        problems [1] SET OF SEQUENCE {
                                            problem [0] INTEGER,
                                            type [1] AttributeType,
                                            value [2] AttributeValue
                                                        OPTIONAL } }
```

```
NameErrorParameter          ::=    SET {
                                       problem [0] INTEGER,
                                       matched [1] Name }

ServiceErrorParameter       ::=    SET {
                                       problem [0] INTEGER }

ReferralErrorParameter      ::=    SET {
                                       candidate [0] ContinuationReference }

SecurityErrorParameter      ::=    SET {
                                       problem [0] INTEGER }
```

Appendix B - The ELROS Data Structures Corresponding To The X.500 Fragment of Appendix A

```
/* From X.208 */

    VisibleString(16) <universal 23 implicit> UTCTime

/* From X.501 */

    ObjectIdentifier(20,4) AttributeType;

    Any AttributeValue;

    Record <sequence>
        {
        AttributeType type;
        DynamicArray <set> AttributeValue values;
        } Attribute;

    Record <sequence>
        {
        AttributeType attributeType;
        AttributeValue attributeValue;
        } AttributeValueAssertion;

    DynamicArray <set> AttributeValueAssertion RelativeDistinguishedName;

    DynamicArray RelativeDistinguishedName RDNSequence;

    Union
        {
        RDNSequence rDNSequence;
        } Name;

    RDNSequence DistinguishedName;
```

```
/* From X.509 */

    Record <sequence>
        {
        ObjectIdentifier(20,4) algorithm;
        Any <null> parameters;
        }  AlgorithmIdentifier;

    Record <sequence>
        {
        UTCTime notBefore;
        UTCTime notAfter;
        } Validity;

    Record <sequence>
        {
        AlgorithmIdentifier  algorithm;
        BitString(1024) subjectPublicKey;
        } SubjectPublicKeyInfo;

    Record <sequence>
        {
        Record <sequence>
            {
            Integer(4) <context 0 null 0 default 0> version;
            Integer(8)  serialNumber;
            AlgorithmIdentifier  signature;
            Name  issuer;
            Validity validity;
            Name  subject;
            SubjectPublicKeyInfo subjectPublicKeyInfo;
            } CertificateBody;
        AlgorithmIdentifier  signatureAlgorithm;
        BitString(2048) signatureValue;
        } Certificate;
```

```
    Record <sequence>
        {
        Certificate <context 0 null> forward;
        Certificate <context 1 null> reverse;
        } CertificatePair;

    Record <sequence>
        {
        Certificate userCertificate;
        DynamicArray <null> CertificatePair theCACertificates;
        } CertificationPath;


/* From X.520 */

    Record <sequence>
        {
        OctetString(64) <context 0 null> pSelector;
        OctetString(64) <context 1 null> sSelector;
        OctetString(64) <context 2 null> tSelector;
         DynamicArray <null context 3 implicit set> OctetString(64) nAddresses;
        } PresentationAddress;

/* From X.518 */

    Record
        {
        Enumerated(1) <context 0> nameResolutionPhase;
        Integer(1) <context 1 null> nextRDNToBeResolved;
        } OperationProgress;

    Enumerated(1) ReferenceType;

    Record
        {
        Name <context 0> ae-title;
```

```
        PresentationAddress <context 1> address;
        }  AccessPoint;

    Record
        {
        Name <context 0> targetObject;
        Integer(4) <context 1 null> aliasedRDNs;
        OperationProgress <context 2> operationProgress;
        Integer(4) <context 3 null> rdnsResolved;
        ReferenceType <context 4 null> referenceType;
        DynamicArray <context 5 set> AccessPoint accessPoints;
        }  ContinuationReference;

/* From X.511 */

    Record
        {
        Union <null>
                {
                Null <context 0> allAttributes;
                /* the default for AttributeType must be implemented
                        by the programmer in executable code */
                DynamicArray <context 1 set> AttributeType select;
                } attributeTypes;
        Integer(4) <context 2 null default 1> InfoTypes;
        }  EntryInformationSelection;

     Union
        {
        AttributeType attributeType;
        Attribute attribute;
        }  AVA;

    Record <sequence>
        {
        DistinguishedName  distinguishedName;
```

```
     Boolean(1) <null true default true> fromEntry;
     DynamicArray <set null> AVA ava;
     } EntryInformation;

Record
     {
     /* the default for options (a null BitString) must be implemented
             by the programmer in executable code */
     BitString(64) <context 0 null > options;
     Integer(4) <context 1 null 1 default 1> priority;
     Integer(4) <context 2 null> timeLimit;
     Integer(4) <context 3 null> sizeLimit;
     Integer(4) <context 4 null> scopeOfReferal;
     } ServiceControls;

Integer(1) ProtectionRequest;

Record
     {
     CertificationPath <context 0 null> certification-path;
     DistinguishedName <context 1 null> name;
     UTCTime <context 2 null> time;
     BitString(256) <context 3 null> random;
     ProtectionRequest <context 4 null> target;
     } SecurityParameters;

Record
     {
     Integer(4) <context 0> identifier;
     Boolean(1) <context 1 null false default false> critical;
     Any <context 2> item;
     } Extension;

Record
     {
     /* the default for selection and serviceControls (a null Set) must be
```

```
        implemented by the programmer in executable code */
    Name <context 0> object;
    EntryInformationSelection <context 1 null> selection;
    ServiceControls <context 30 null> serviceControls;
    SecurityParameters <context 29 null> securityParameters;
    DistinguishedName <context 28 null> requestor;
    /* the default for operationProgress (a Set containing an enumerated
        element with the value notStarted) must be implemented by the
        programmer in executable code */
    OperationProgress <context 27 null> operationProgress;
    Integer(4) <context 26 null 0> aliasedRDNs;
    DynamicArray <context 25> Extension extensions;
    } ReadArgumentType;

Union
    {
    ReadArgumentType  readArgument;
    Record <sequence>
        {
        ReadArgumentType  readArgument;
        AlgorithmIdentifier  algorithmIdentifier;
        BitString(2048) signature;
        } signatureBlock;
    } ReadArgument;

Record
    {
    EntryInformation <context 0> entry;
    SecurityParameters <context 30 null> securityParameters;
    DistinguishedName <context 29 null> performer;
    Boolean(1) <context 28 null false default false> aliasDereferenced;
    } ReadResultType;

Union
    {
    ReadResultType readResult;
```

```
    Record <sequence>
            {
            ReadResultType readResult;
            AlgorithmIdentifier  algorithmIdentifier;
            BitString(2048) signature;
            } signatureBlock;
    } ReadResult;

Record <sequence>
    {
    Integer(4) <context 0> problem;
    AttributeType <context 1> type;
    AttributeValue <context 2 null> value;
    } Problems;

Record
    {
    Name <context 0> object;
    DynamicArray <context 1> Problems problems;
    } AttributeErrorParameter;

Record
    {
    Integer(4) <context 0> problem;
    Name <context 1> matched;
    } NameErrorParameter;

Record
    {
    Integer(4) <context 0> problem;
    } ServiceErrorParameter;

Record
    {
    ContinuationReference <context 0> candidate;
    } ReferralErrorParameter;
```

```
Record
    {
    Integer(4) <context 0> problem;
    } SecurityErrorParameter;
```

Appendix C - The Syntax of Basic Type Constants

Syntax for constants of the various ELROS basic types is rigorously defined here (e is the empty word) :

under ::= { **_** }
dot ::= { **.** }
bit ::= { **0**, **1** }
ubit ::= <bit> | <under>
digit ::= { **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9** }
udigit ::= <digit> | <under>
uhexdigit ::= <digit> | { **A**, **B**, **C**, **D**, **E**, **F**, **a**, **b**, **c**, **d**, **e**, **f** }
signopt ::= { **+**, **-** } | e

Boolean ::= { **true**, **false** }
Integer ::= <signopt> <udigit>$^+$
BitString ::= <ubit>$^+$
ObjectIdentifier ::= <digit>+ (<dot> <digit>$^+$)$^+$
Real ::= <digit>* <dot> <digit>* ({ **E**, **e** } <Integer>)?
OctetString ::= **0x** <uhexdigit>$^+$
GeneralString (et. al) ::= *identical to C-string syntax*

Index