IDL^4 Version 1.0.0
User's Manual

Andreas Haeberlen
University of Karlsruhe
haeberlen@ira.uka.de

April 2003

Preface

This manual is still under construction. More information on the technology used in IDL[4] can be found in [3]; for more details on CORBA IDL, please read [1] or one of the countless books on CORBA, e.g. [7].

Contents

Chapter 1

Writing interface definitions

1.1 Basic structure

The specification in an IDL file describes one or more interfaces, which in turn may contain one or more methods. Here is an example:

```
module storage {
   interface textfile {
      void readln(
        inout short pos,
        out string line
      );
      void writeln(
        inout short pos,
        in string line
      );
      int get_pos();
   };
};
```

```
library storage {
   interface textfile {
      void readln(
        [in, out] short *pos,
        [out, string] char **line
      );
      void writeln(
        [in, out] short *pos,
        [in, string] char **line
      );
      int get_pos();
   };
};
```

IDL[4] supports two specification languages, CORBA IDL and DCE IDL, so most example code is shown in both languages; the CORBA code is always on the left side.

Note that the syntax is very similar to C/C++, especially in the case of DCE IDL. The most important difference is that every parameter has a directional attribute (`in`, `inout` or `out`). This is used to indicate whether the parameter contains

- input data, which must be copied from client to server,
- output data, which is returned by the server, or
- a combination of both

Another difference is the presence of special data types like `string`, which have additional semantics; for example, `string` stands for a sequence of characters with a trailing zero, and `sequence` denotes an array of variable length. These special types are explained later in this chapter.

Finally, it is possible to avoid naming conflicts by putting interfaces into modules. For example, a network driver and a postal application might both want to provide a `packet` interface, possibly with different methods; this conflict can be resolved by using two separate modules.

1.2 Types

1.2.1 Basic types

Integers

IDL^4 supports the following CORBA integer types:

Type name	Size	Value range
unsigned short	16 bit	0..65535
short	16 bit	-32768..32767
unsigned long	32 bit	0..4294967295
long	32 bit	-2147483648..2147483647
unsigned long long	64 bit	0..18446744073709551615
long long	64 bit	-9223372036854775808..9223372036854775807

The type `int` is also supported, although it is deprecated because the standard does not define its size (32 or 64 bit). Currently, it has 32 bit on all platforms.

Floating point

The following floating point types are available:

Type name	Size	Value range	Precision
float	32 bit	$3.403 \cdot 10^{38}$	6 digits
double	64 bit	$1.798 \cdot 10^{308}$	15 digits
long double	80 bit	$1.1897 \cdot 10^{4932}$	18 digits

Characters

IDL^4 supports both the 8-bit `char` and the 16-bit `wchar` data types. The type `unsigned char` may also be used, but it is deprecated. The reason is that in CORBA IDL, characters have special semantics; for example, they might be translated to a different character set by the marshalling code.

If you need an 8-bit data type for binary data, you should use the `octet` type.

Flexpages

IDL^4 supports the L4 mapping primitives by providing a special type named `fpage`. This type corresponds to the flexpage type of the respective kernel interface; its size is platform-dependent.

Miscellaneous

Type name	Size	Possible values
boolean	1 bit	`true`, `false`
octet	8 bit	0..255
void	undefined	none

The `void` data type may only be used for return values or as the base type for a pointer. The `Object` and `any` types, which are also defined by CORBA, are not supported.

1.2.2 Constructed types

Alias types

You can use `typedef` to create your own types:

```
typedef unsigned short word_t;
typedef string<40> max40char_t;
typedef long array_t[4][3];
```

```
typedef unsigned short word_t;
/* Bounded strings not supported */
typedef long array_t[4][3];
```

IDL[4] maps the types in an interface description to the target language and adds a definition to the header files it produces. Thus, you can also use these types in your own code.

Sequences

CORBA provides a special type for transferring variable-length data, the `sequence` type. A sequence has a base type (e.g. `char`) and, optionally, a size bound. Consider the following example:

```
typedef sequence<float, 7> some_t;
typedef sequence<char> another_t;
```

```
/* DCE IDL does not support
   the sequence type */
```

The first line defines an array of `float`s that does not contain more than seven members (but may contain less); the second line defines a character array of arbitrary size. Note that sequences may only appear in `typedef`s, not directly as an argument type.

When using sequences, please consider that IDL[4] needs to preallocate buffer space for them. Providing a tight size bound saves memory and considerably improves performance.

Note that the sequence mapping in IDL[4] differs from the one specified in [2]. In the IDL[4] mapping, the programmer is *always* responsible for the storage allocated for output sequences; the release flag is not supported. Also, output sequence parameters do not use double indirection; instead, they are treated just like ordinary structs.

Arrays

In DCE, there is no single type for variable-length arguments. Instead, size and location of the data are specified independently. Consider the following example:

```
/* CORBA does not support
   the length_is attribute */
```

```
interface foo {
  void bar(
    [in] int len,
    [in, length_is(len)] float *addr
  );
  void xyz(
    [in, length_is(5)] short *addr
  );
};
```

The stub code for `bar` transfers `len` floating point numbers, starting at `addr`, whereas the code for `xyz` always transfers five 16-bit integers.

On the server side, buffer space is allocated and freed by the stub code. In particular, for `out` arguments, the stub preallocates enough buffer space and passes a pointer to it when the function is invoked. Nevertheless, your code is not required to use this buffer; it can save one copy operation by returning a pointer of its own.

On the client side, the stub allocates buffers for output values, but does not free them. It is your responsibility to invoke `CORBA_free()` for every array that is returned by the server.

Note that the argument to `length_is` denotes the number of elements, *not* the size in bytes!

Structs

Structs are used exactly like their counterparts in C/C++:

```
struct some_t {
  short a[4], b;
  float c;
};
```

```
struct some_t {
  short a[4], b;
  float c;
};
```

The stub allocates and frees buffers for the server; on the client side, this is the responsibility of the user. However, `CORBA_alloc()` is not used because structs have a fixed size; instead, for an `out` parameter, the user supplies a pointer to an existing struct, which is then overwritten by the stub.

Unions

Unlike C/C++-style unions, a CORBA-compliant union needs a special member, the discriminant, which is used to decide which type the union currently contains. This is important because different types are usually marshalled differently. Consider the following example:

```
union U switch (int) {
  case 1 : long x;
  case 2 :
  case 3 : string s;
  default: char c;
};
```

```
union U switch (int a) {
  case 1 : long x;
  case 2 :
  case 3 : string s;
  default: char c;
};
```

If the discriminant, which is always named `_d` in CORBA, has the value 1, then one 32-bit value needs to be copied. When `_d` is 2 or 3, however, the union contains a pointer, which must be dereferenced, resulting in a string to be transferred.

Note: Unions are not yet available in the current IDL[4] *release!* However, C/C++-style unions are supported. Such a union is always copied directly and entirely; thus, it may not contain types with special semantics, such as strings.

Strings

CORBA supports 8-bit and 16-bit strings with an optional length bound; strings are always terminated by the value zero. Consider the following example:

```
typedef string<20> s20_t;

interface foo {
  void bar(
    in string a,
    inout s20_t b,
    out wstring<40> c
  );
};
```

```
/* No bounded strings in DCE */

interface foo {
  void bar(
    [in, string] char *a,
    [in, out, string] char **b,
    [out, string] short **c
  );
};
```

The first argument to `foo::bar` is an 8-bit string of arbitrary length, whereas the second argument may contain at most 20 characters; the third argument is a 16-bit string of not more than 40 elements. The first two arguments are terminated by a zero byte, the third one ends with a 16-bit zero value.

On the server side, strings are managed by the stubs. For output values, sufficient buffer space is allocated before the method is invoked; however, the implementation is free to move the pointer to another buffer. On the client side, the stubs allocate output buffers using `CORBA_alloc()`, but do not free them; it is the responsibility of the user to invoke `CORBA_free()` for each one.

Fixed point

CORBA includes a special type for fixed-point, BCD-coded numbers. This type is not supported by IDL^4 .

Bitfields

IDL^4 supports C/C++-style bitfields. These are allowed neither by CORBA nor by DCE and should be used with care, because they are highly platform-dependent. Consider the following example:

```
struct msgdope_t {
  long cc     : 8;
  long parts  : 5;
  long mwords : 19;
};
```

```
struct msgdope_t {
  long cc     : 8;
  long parts  : 5;
  long mwords : 19;
};
```

1.3 Exceptions

Currently, only the CORBA exception handling is supported by IDL^4 . In CORBA, there are two classes of exceptions: system and user-defined. System exceptions may be raised by any method (e.g. when the IPC fails or a timeout happens), whereas user-defined exceptions must be explicitly specified by the interface description. Consider the following example:

```
interface file {
  exception access_denied {};
  exception not_found {};

  void open(in string name)
    raises (access_denied,
            not_found);
};
```

```
/* DCE exception handling
   not supported */
```

This means that the method `file::open` can, in addition to system exceptions, raise two user-defined exceptions named `access_denied` and `not_found`.

CORBA also allows exceptions to contain additional information; for example, it may be useful to add a message for the user, or details on how to correct the error. Here is an example:

```
interface file {
  exception access_denied {
    string reason;
    int missing_privileges;
  }

  void open(in string name)
    raises (access_denied);
};
```

```
/* DCE exception handling
   not supported */
```

Note: Currently, IDL^4 only supports exceptions without additional information

1.4 Constants

It is possible to define constants within an interface. The constants are added to the generated header files, but may also be used within the specification itself. Consider the following example:

```
interface foo {
  const int addr_size = 6;
  struct hwaddr {
    octet mac[addr_size];
  };
};
```

```
interface foo {
  const int addr_size = 6;
  struct hwaddr {
    unsigned small mac[addr_size];
  };
};
```

This causes a constant named `addr_size` to be exported to the client header file; also, the struct `hwaddr` is declared to be six bytes long.

1.5 Attributes

1.5.1 Oneway functions

Usually, a remote procedure call consists of two phases: A send phase, in which the client sends a message to the server, and a receive phase, in which it waits for a reply. However, in some cases, it is necessary to omit the second phase, e.g. when the request is to be processed asynchronously, or when no reply is possible.

Here is how this behaviour can be specified:

```
interface foo {
  oneway void bar(in int a);
};
```

```
interface foo {
  [oneway] void bar([in] int a);
};
```

Oneway methods must not have `inout` or `out` arguments; also, they may neither return a value nor raise any exceptions. Note that when the method returns on the client side, the absence of exceptions does not mean that the request has been processed successfully; it only indicates that the request transfer did not fail.

1.5.2 Function identifiers

As interfaces can contain multiple methods, and servers may implement multiple interfaces, the server must be able to tell from the request which method it is intended for. In IDL[4] , this is accomplished with numeric function IDs.

A function ID has two parts: An interface number and a method number. The interface number is identical for all methods in an interface, whereas different interfaces may be assigned the same number. The method number must be unique within an interface.

By default, IDL[4] assigns the number 0 to all interfaces; this implicitly assumes that different interfaces are implemented by different threads. If this is not the case, you need to assign the interface numbers manually. The allowed range for interface numbers is platform dependent; typically, numbers of up to 1.000 are supported. Here is an example:

```
[uuid(5)]
interface foo {
  void bar(in int a);
};
```

```
[uuid(5)]
interface foo {
  void bar([in] int a);
};
```

You can also change the assignment of method numbers by applying the `uuid` attribute to individual methods. However, this is rarely necessary.

1.5.3 Mapping fpages

The L4 IPC primitive supports mapping, which is essentially the transfer of a complete memory page from one address space to another. As a result, the page is shared by both address spaces; write operations in either of them are instantly visible in both. Also, the page may cover the same address range in both spaces, but this is not mandatory.

IDL^4 supports this primitive with a special data type, the `fpage`, which describes a memory region; see [4, 5, 8] for more details. `fpage`s are implicitly mapped during message transfer; the mapping persists and is not revoked upon completion of the call. Consider the following example:

```
interface pager {
  [uuid(0)]
  void pagefault(
    in int addr, in int ip,
    out fpage f
  );
};
```

```
interface pager {
  [uuid(0)]
  void pagefault(
    [in] long addr, [in] long ip,
    [out] fpage *f
  );
};
```

This defines a method `pagefault` which takes two arguments, the fault address `addr` and the instruction pointer `ip`; the server replies with an fpage `f` which is to be mapped to the client address space.

1.6 Advanced features

1.6.1 Local functions

Some L4 microkernels support a special IPC primitive, the local IPC or lipc, which is optimized for intra-address space calls [6]. This feature can be used with IDL^4 as follows:

```
local interface foo {
  short bar(
    in short a,
    in short b
  );
};
```

```
[local] interface foox {
  short bar(
    [in] short a,
    [in] short b
  );
};
```

The `local` attribute indicates that the methods in this interface will *only* be called via lipc. This permits IDL^4 to apply considerably more optimizations; for example, lipc will be used by both client and server stubs, and parameters may be passed by reference. Possible applications include semaphore servers or dispatcher threads, which are used for distributing incoming requests to multiple worker threads in the same address space.

Note: This is an experimental feature in the current release!

1.6.2 Type import from C++ code

Usually, interface specifications should contain definitions for the data types they use. However, it may sometimes be convenient to import additional data types from the application code. IDL^4 supports this with a DCE-style `import` statement:

```
interface foo {
  import "l4/x86/types.h";

  void bar(in l4_taskid_t tid);
};
```

```
interface foo {
  import "l4/x86/types.h";

  void bar([in] l4_taskid_t tid);
};
```

This makes available the types defined in `l4/x86/types.h` to all the methods in interface `foo`; IDL^4 contains a gcc-compatible C++ parser for this purpose. At compile time, the header file is scanned, all global type definitions are imported and converted into IDL.

Note that unlike types defined directly in IDL, the generated header files do not contain definitions for imported types; instead, an `#include` directive referring to the original file is added.

1.6.3 Disabling the memory allocator

By default, IDL^4 uses CORBA-style dynamic memory allocation, i.e. it calls `CORBA_alloc` to reserve buffers for variable-length output values. In some cases, however, the data is expected at a specific location, which requires an additional copy on the client side.

To avoid this problem, you can use the `prealloc` attribute. Consider the following example:

```
interface foo {
  typedef sequence<long, 100> buf;
  void bar(
    [prealloc] out buf x
  );
};
```

```
interface foo {
  void bar(
    [out, prealloc, length_is(len)]
    char **data, [out] short *len
  );
};
```

No dynamic buffers are allocated in either case. Instead, you must explicitly provide a buffer by initializing `x._buffer` or `*data`, respectively. Also, you must supply the size of the buffer in `x._maximum` (or `*len` in the DCE example).

While `prealloc` should be used for individual arguments, a command line option is also available which disables dynamic allocation for the entire file.

1.6.4 Receiving kernel messages

In the L4 world, exceptions are mapped to IPCs, which are sent by the kernel on behalf of the faulting application, e.g. to its pager. These messages can be received by IDL^4 stubs when special interface definitions are used.

The following interface handles X0-style page faults, which are e.g. generated by the Hazelnut kernel:

```
interface pager {
  [uuid(0)] void pagefault(
    in long addr,
    in long ip,
    out fpage p
  );
};
```

```
interface pager {
  [uuid(0)] void pagefault(
    [in] long addr,
    [in] long ip,
    [out] fpage *p
  );
};
```

Here is how a V4 page fault is specified (e.g. for the Pistachio kernel):

```
interface pager {
  [kernelmsg(idl4::pagefault)]
  void pagefault(
    in long addr,
    in long ip,
    in long privileges,
    out fpage p
  );
};
```

```
interface pager {
  [kernelmsg(idl4::pagefault)]
  void pagefault(
    [in] long addr,
    [in] long ip,
    [in] long privileges,
    [out] fpage *p
  );
};
```

Note that in this case, the fault type (i.e. the requested privileges) is provided as a separate argument, whereas in the upper example, it is encoded in the lower two bits of the fault address.

Chapter 2

Working with generated code

2.1 Compiling the IDL file

IDL^4 can generate three types of output from a given interface description:

- *Client stubs*, which are linked with every client application, i.e. every application that needs to invoke methods from the interface
- *Server stubs*, which are used by every server that needs to implement the interface.
- *Server templates*, which essentially contain a dummy implementation for the interface. They can be used as a starting point for writing a server.

Usually, the first two kinds of output are generated during the compilation process, whereas the third is only generated once and then extended with user code. You can select the kind of output by supplying `-c`, `-s` or `-t` on the command line, respectively. For example,

```
idl4 -ix0 -pia32 -ffastcall -s pager.idl -h pager-server.h
```

generates a header file called `pager-server.h` which contains all the server stubs for methods in `pager.idl`. Also, the X0 backend for the IA32 platform is selected, e.g. because the application will use the Hazelnut kernel and run on a Pentium-III processor. Finally, the `fastcall` option is given, which allows IDL^4 to use nonstandard system calls, in this case, the `sysenter` instruction.

2.2 Sending requests

As specified by the CORBA C language mapping [2], client stubs have two implicit parameters (i.e. parameters that are not defined by the interface). Consider the following example:

```
module storage {
   interface textfile {
      void readln(
        inout short pos,
        out string line
      );
   };
};
```

```
library storage {
   interface textfile {
      void readln(
        [in, out] short *pos,
        [out, string] char **line
      );
   };
};
```

When this definition is compiled with the `-c` option, IDL^4 creates the following client stub:

```
void storage_textfile_readln(
        storage_textfile _service,
        CORBA_short *pos, CORBA_char **line,
        CORBA_Environment *_env
)
```

The first parameter, `_service`, contains the thread ID of the server where the request is to be sent. Unlike other CORBA code generators, IDL^4 does not provide a way to obtain this automatically; it assumes that this functionality is implemented by your system.

The last parameter is a pointer to a `CORBA_Environment` structure. This structure contains additional information related to the call, such as a timeout value or a memory window for receiving mappings. You must initialize this structure before invoking the call.

An invocation of `readln` could look like this:

```
#include <storage_client.h>
void test(l4_threadid_t server) {
   CORBA_Environment env = idl4_default_environment;
   short pos = 100;
   char *buf;

   storage_textfile_readln(server, &pos, &buf, &env);

   switch (env._major) {
      case CORBA_SYSTEM_EXCEPTION:
         printf("IPC failed, code %d\n",
            CORBA_exception_id(&env));
         CORBA_exception_free(&env);
         return -1;
      case CORBA_USER_EXCEPTION:
         printf("User-defined exception");
         CORBA_exception_free(&env);
         return -1;
      case CORBA_NO_EXCEPTION:
         break;
   }

   printf("Read: %s\n", buf);
   CORBA_free(buf);
}
```

Note how `env` is initialized with an IDL^4 -supplied default value, which means a timeout of infinity and an empty receive window. Also, the example shows how environment structure can be used to determine whether an exception occurred during the call, and what type it was. Finally, it is important to always release storage allocated by the stubs (e.g. exceptions, output strings, ...) with the appropriate function.

2.3 Processing requests

The standard server loop, which is included in the server template, mainly consists of two macros:

- `idl4_reply_and_wait`, which sends any pending replies and then receives one incoming request, and

- `idl4_process_request`, which analyzes the request and calls the appropriate implementation function

Between those macros, you can insert additional code, e. g. a permission check. The second macro uses a function table to decide which function should handle the request; it takes the method number as an argument and uses it as an index into the table. Table entries which correspond to unassigned method numbers contain a reference to the `discard` function of the interface; thus, this function is only called when a malformed request was received.

The server template file also contains function templates for every method in the interface. For the example interface from above, the following template is created:

```
IDL4_INLINE void storage_textfile_readln_implementation(
   CORBA_Object _caller, CORBA_short *pos,
   CORBA_char **line, idl4_server_environment *_env
)

{
   /* implementation of storage::textfile::readln */

   return;
}

IDL4_PUBLISH_STORAGE_TEXTFILE_READLN(
   storage_textfile_readln_implementation
);
```

Similar to the client side, the function has two additional parameters. The first parameter, `_caller`, contains the ID of the thread that has sent the request, whereas the last parameter, `_env`, points to an internal data structure. Many functions in the IDL[4] runtime need a pointer to this structure, for example the function `CORBA_exception_set`, which is used to raise exceptions.

Please note the macro at the end of the function. This macro makes the function accessible to the server loop; for the optimizations to work, it is also very important that this macro is included *after* the implementation function.

Unlike the client side, memory allocation on the server side is mostly handled by the stub code, that is, you do not need to call `CORBA_free`, except if you explicitly allocated additional buffers. The stub code preallocates a buffer for every `out` value and passes a pointer to the implementation function in the respective argument. However, for large buffers (such as strings), you can save one copy by overwriting the pointer with another one pointing directly to the data you want to send.

If you decide not to send a reply at all, you can use the `idl4_set_no_response` function. In this case, the stub code will discard any `inout` or `out` values and skip the send phase; instead, it will directly receive another request.

Chapter 3

Quick reference guide

3.1 Invoking IDL^4

IDL^4 is generally invoked in the following way:

```
idl4 [OPTIONS] input.idl
```

The following sections contain a more detailed description of the individual options.

3.1.1 Overall Options

`-c`
`-s`
`-t`

These options are used to control the kind of output. The following output types are available:

- *Client stubs (-c)* are used by client tasks to invoke a service
- *Server stubs (-s)* contain functions to receive and dispatch incoming requests on the server side
- *Server templates (-t)* can be used as a starting point when implementing a new service. They contain function prototypes and a simple server loop.

`-o` *`filename.c`*

Place code output in *filename.c*. Note that for some mappings, the stub code is contained entirely in the header files.

`-h` *`filename.h`*

Create a header file called *filename.h*. This option cannot be used together with `-t`, because server templates do not include a header file.

`-v`

Print the version number and build date on standard output.

3.1.2 Warning Options

`-Wprealloc`

Warn whenever a stub needs to allocate an excessive amount of heap memory, e.g. when an unlimited output string is encountered.

`-Wall`

All of the above '-W' options combined.

3.1.3 Debugging Options

`-dtest`	Generate test code. When used with `-t`, this creates a test application that can run as a root task on the bare microkernel. The code creates a server for each interface in the IDL and then invokes every operation with random parameters.
`-dparanoid`	Add even more checks to the test code. For example, test for stack overflow and monitor the message dopes.
`-dcpp`	Print each input line as it is read from the C preprocessor.
`-daoi`	Dump the abstract syntax tree generated from the IDL file.
`-dcheck`	Generate debug output for the semantical analysis and consistency check.
`-dcast`	Dump the C abstract syntax tree.
`-dfids`	Print the function ID assignment.
`-dmarshal`	Show details about the marshalling stage.
`-dgenerator`	Analyze the stub generator and the backend.
`-dimport`	Generate debug info while importing C++ headers.
`-dreorder`	Show details when reordering parameters.
`-dvisual`	Add debug output to the server stubs.
`-d`	All of the above.

3.1.4 Miscellaneous Options

`-ffastcall`	Allows IDL4 to use a non-conforming IPC system call where available. The resulting code is faster, but may not be entirely compliant with the kernel specification. For example, the X0/IA32 backend may use `sysenter` to invoke the IPC system call.
`-flipc`	Enables support for local IPC on X0-style kernels.
`-fctypes`	Use ANSI C types instead of the platform-independent CORBA types (like `CORBA_long`).
`-fomit-frame-pointer`	Assume the frame pointer is not used. This option must be consistent with the corresponding gcc option.
`-fuse-malloc`	Allows IDL4 to use dynamic memory allocation (e.g. `CORBA_alloc`). This is enabled by default; if you disable it, you must preallocate buffers for *all* variable-sized output parameters, such as `sequence` or `string`.
`-fomit-empty-files`	Prevents files from being generated if they do not contain any code. This is enabled by default.
`-floop-only`	In server templates, removes all definitions except server loops.
`-fmodules-only`	In server templates, generate code only for interfaces that are defined in a module.

These options can also be disabled by using a `no-` prefix (e.g. `-fno-fastcall`).

3.1.5 Preprocessor Options

`-D` *`macro`*	Defines *macro* in the preprocessor.
`-I` *`path`*	Adds *path* to the current include path. The include path is passed to the preprocessor and is also searched when importing types.
`-Wp,`*`option`*	Passes *option* directly to the preprocessor.
`--with-cpp` *`path`*	Use another preprocessor (the default is /usr/bin/cpp)

3.1.6 Target Options

`-m` *`mapping`*	When generating stub code, use the specified mapping (C or C++).
`-i` *`interface`*	Produce code for the specified kernel interface.
`-p` *`platform`*	Optimize for the specified platform (`generic` is allowed).
`-w` *`wordsize`*	Specifies the word size in bits (32 or 64) where it cannot be determined from the platform, e.g. in the generic backends.
`--sys-prefix` *`path`*	Add a prefix to all system include files

Chapter 4

Appendix

4.1 Keywords

The following strings are reserved as keywords and should not be used for identifiers:

all_caches	float	nocache	switch
any	gather	noxfer	TRUE
attribute	import	Object	typedef
base_is	in	octet	union
boolean	include	oneway	unsigned
case	inout	out	uuid
char	int	raises	void
const	interface	readonly	wchar
context	l1_only	ref	writeable
default	length_is	scatter	wstring
double	library	sequence	
enum	local	short	
exception	long	size_is	
FALSE	max_size	string	
fixed	module	struct	

4.2 Supported platforms

This table shows the current status of the individual IDL[4] backends. You can select one of these backends with the `-i` and `-p` options:

Platform	Interface	Current status
IA32	X0	Supported
	V2	Supported; no mapping
	V4	Supported
Generic	X0	Supported
	V2	Supported; no mapping
	V4	Supported

4.3 Supported types

Basic data types

short	signed 16 bit integer	supported
long	signed 32 bit integer	supported
long long	signed 64 bit integer	supported
unsigned short	unsigned 16 bit integer	supported
unsigned long	unsigned 32 bit integer	supported
unsigned long long	unsigned 64 bit integer	supported
float	floating point (32 bits)	supported
double	floating point (64 bits)	supported
long double	floating point (80 bits)	supported
char	8 bit character	supported
wchar	16 bit character	supported
boolean	boolean value	supported
any	wildcard type	-
octet	unsigned 8 bit integer	supported
enum	enumeration	supported
ref	pointer	supported
fpage	memory flexpage	supported
fixed	fixed-point decimal	-

Constructed Types

string	unbounded string	supported
string<n>	bounded string	supported
wstring	unbounded wide string	supported
sequence<type>	type sequence	supported
sequence<type,n>	bounded type sequence	supported
struct	structure	flat members supported
union	union	DCE style supported
Object	object reference	supported
array	array	supported
typedef	alias type	supported
exception	exception	memberless exceptions supported

Bibliography

[1] Object Management Group. CORBA 2.6 specification. http://www.omg.org/cgi-bin/doc?formal/01-12-35.

[2] Object Management Group. CORBA C language mapping. http://www.omg.org/cgi-bin/doc?formal/99-07-35.

[3] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *Proceedings of the First Workshop on Industrial Experiences with Systems Software (WIESS)*, pages 31–38, San Diego, CA, October 2000.

[4] Jochen Liedtke. Lava nucleus reference manual. http://i30www.ira.uka.de/publications/pub1998/ln-86-21.ps, March 1998.

[5] Jochen Liedtke. L4 nucleus version X.0 reference manual. http://l4ka.org/documentation/files/l4-86-x0.ps, September 1999.

[6] Jochen Liedtke and Horst Wenske. Lazy process switching. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.

[7] Jens-Peter Redlich. *CORBA 2.0*. Addison-Wesley, 1996.

[8] The L4Ka Team. L4 X.2 reference manual. http://l4ka.org/documentation/files/l4-x2.pdf, January 2002.