

DICE Version 3.1.1
User's Manual

Ronald Aigner
Technische Universität Dresden, Germany
Ronald.Aigner@os.inf.tu-dresden.de

January 19, 2007

Preface

This documentation is still under heavy construction.

Contents

1	Interface Definitions	7
1.1	A Simple Interface	8
1.1.1	Interface Description Language	8
1.2	Supported Types	8
1.2.1	Integer Types	8
1.2.2	Floating Point Types	9
1.2.3	Other Types	10
1.2.4	L4 specific types	10
1.3	Constructed Types	10
1.3.1	Aliased Types	11
1.3.2	Arrays and Sequences	11
1.3.3	Structured Types	13
1.3.4	Unions	14
1.4	Deriving Interfaces	14
1.5	Constants	15
1.6	Attributes	15
1.6.1	Array Attributes	16
1.6.2	Strings	17
1.6.3	Indirect Parts IPC	17
1.6.4	Receiving Indirect Parts	18
1.6.5	Error Function	19
1.6.6	Using other types for transmission	19
1.7	Imported and Included Files	20
1.7.1	C/C++ Attributes	20
1.8	Message Passing	20
1.8.1	Optimizations	21
1.9	Asynchronous Servers	22
1.10	Thoughts About DICE Specific Types	23
1.10.1	flexpage	23

1.10.2	Interrupts	24
2	Using the Generated Code	25
2.1	Invoking DICE	25
2.2	Client Code	26
2.3	Server Code	26
2.4	Server Loop	27
2.5	Function Identifiers — Opcodes	28
2.5.1	Defining Opcodes	29
2.6	CORBA Environment	31
2.6.1	CORBA Environment Functions	32
2.6.2	DICE Specific Extensions	33
2.7	Buffer Management	34
2.8	Memory Management for Indirect Parts	34
2.8.1	malloc and CORBA's alloc	34
2.8.2	init rcvstring	35
3	A Short Reference to Compiler Options	37
3.1	Compiler Options	37
3.1.1	Pre-Processing Options (Front-End Options)	37
3.1.2	Back-End Options	39
3.1.3	General Options	42
3.1.4	Compiler Flags	42
3.2	Warnings	47
A	Pagefault Handling	49
A.1	L4 version 2	49
A.2	L4 version X.0	50
B	Pagefault Handler	51
C	Message Buffer Access Macros	53
D	CORBA Environment	57
D.1	L4 version 2 and experimental version X.0	57
D.2	L4 experimental version X.2	58
D.3	Linux sockets	59

Chapter 1

Interface Definitions

The use of IDL compilers is well established in distributed systems. An IDL compiler provides you with a level of abstraction for the communication with a remote service.

Assuming you want to write a server for L4 manually, you have to solve several problems. You have to communicate your request from a client to the server, which means that you build a message and send an IPC containing this message to the server. The server has to wait for request, extract the data from the message, and dispatches the request to the appropriate function, which does the requested work. After this function returns, the server has to pack the response into another message and reply to the client.

With the help of an IDL compiler you can automate all of the above steps—except the implementation of the server function. All you have to do is to write a description of the server’s interface using an interface description language (IDL) and translate this description with DICE.

Currently, DICE supports two different interface description languages—CORBA IDL and DCE IDL. Since we think that DCE IDL is more powerful, we added options to express L4 specific properties to the DCE IDL. These attributes are described in detail in section [1.6](#).

For the CORBA IDL there exist multiple well-defined language mappings. Since these are widely accepted, DICE generates code that conforms to the CORBA C language mapping [\[1\]](#).

1.1 A Simple Interface

To illustrate the use of DICE we show a simple example here. The language of our IDL examples is DCE IDL. See the appendix for a CORBA IDL version of the example.

1.1.1 Interface Description Language

An interface description file consists of at least an *interface specification* which includes one or more *function declarations*. As described above a client may send data to the server and receive a response containing data. We have to differentiate these different direction of data transfer from each other. Data sent to the server has the attribute `in` and data sent back to the client has the attribute `out`.

```
interface simple
{
    void foo([in] int parameter);
}
```

The above example sends a simple parameter to the server and does not expect return values. Nonetheless, the client continues its work only after it received an (empty) answer from the server. These calls are always synchronous. Asynchronous calls are described in detail in section 1.9.

1.2 Supported Types

1.2.1 Integer Types

The DCE IDL supports the following types:

type	size	value range
[signed] small [int]	8 bit	-128... 128
unsigned small [int]	8 bit	0... 255
[signed] short [int]	16 bit	-32'768... 32'767
unsigned short [int]	16 bit	0... 65'565
signed	32 bit	-2'147'483'648... 2'147'483'647
[signed] int	32 bit	-2'147'483'648... 2'147'483'647
[signed] long [int]	32 bit	-2'147'483'648... 2'147'483'648

next page ...

... continued from last page

unsigned [int]	32 bit	0... 4'294'967'295
unsigned long [int]	32 bit	0... 4'294'967'295
[signed] long long [int]	64 bit	-9'223'372'036'854'775'808... 9'223'372'036'854'775'807
unsigned long long [int]	64 bit	0... 18'446'744'073'709'551'615
[signed] hyper [int]	64 bit	-9'223'372'036'854'775'808... 8'223'372'036'854'775'807
unsigned hyper [int]	64 bit	0... 18'446'744'073'709'551'615

The CORBA IDL supports the following types:

type	size	value range
short	16 bit	-32'768... 32'767
unsigned short	16 bit	0... 65'565
long	32 bit	-2'147'483'648... 2'147'483'648
unsigned long	32 bit	0... 4'294'967'295
long long	64 bit	-9'223'372'036'854'775'808... 9'223'372'036'854'775'807
unsigned long long	64 bit	0... 18'446'744'073'709'551'615

Note that the interface description language do *not* provide a type to describe a machine word. To use parameters of the type machine word, import a header file defining such a type and use this type.

1.2.2 Floating Point Types

The DCE and CORBA IDL support the following types:

type	size
float	32 bit
double	64 bit
long double	80 bit

1.2.3 Other Types

DICE also supports miscellaneous types for DCE IDL:

type	size	values
byte	8 bit	0... 255
void	undefined	none
unsigned char	8 bit	0... 255
[signed] char	8 bit	-128... 127
boolean	8 bit	true, false

The type `boolean` is mapped to a `unsigned char` type and therefore has a size of 8 bit. Its values are `false` if zero and `true` otherwise.

DICE supports the following miscellaneous types for the CORBA IDL:

type	size	values
char	8 bit	-128... 127
wchar	16 bit	-32'768... 32'767
boolean	8 bit	true, false
octet	8 bit	0... 255

The types `char` and `wchar` have a special semantic in the CORBA IDL. Even though they might contains integer values, they are interpreted as characters. The CORBA specification allows the communication code to transform character sets and thereby change the integer value of a character. If you do not intend to transfer string, but rather sequences of 8 Bit values, use the `octet` type instead.

1.2.4 L4 specific types

There are some L4 specific types, which have been added to the IDLs to support L4 specific semantics. One of these types is the flexpage type, which expresses the mapping of memory pages from one address space into another. A flexpage can be transmitted using the `flexpage` (or `fpage`) type.

```
void map([in] fpage page);
```

1.3 Constructed Types

DICE transmits constructed types as well. You may either include or import a type via the mechanisms described in section 1.7 or you define it within

the IDL file. The latter approach has the advantage that you can give hints to the IDL compiler on how to transmit the type. The disadvantage is, that you have to either use the generated data type in your wrapper code or cast your own data type to the generated data type when using the generated stubs.

1.3.1 Aliased Types

You may use `typedef` to define alias names for types, which can be used as types of parameters or members of other constructed types.

```
typedef int buffer50[50];
```

The specified types are provided in the generated header files and can therefore be used in your code.

1.3.2 Arrays and Sequences

You may specify arrays in your IDL files.

Arrays are denoted by brackets, and may contain lower and upper bounds. You may also specify a variable sized array by using an asterisk instead of an empty pair of brackets.

```
typedef int unboundArray[];
typedef int unboundArray2[*];
```

The following example describes an array with 20 elements of type `long`. There are different ways to express this. As mentioned you may specify lower bounds for arrays. The generated stub code will then transmit the data starting with the lower bound of the array (*currently not implemented*).

```
void array1([in] long param[20]);
void array2([in] long param[0..20]);

// currently not supported
void array3([in] long param[10..30]);
```

The following example shows some possibilities to specify variable sized arrays. Variable sized arrays may have a fixed size, but the number of transmitted elements is determined at run-time. (*The shown examples will not work, because the `size_is` attribute is missing.*)

```
void var_array1([in] long param[*]);
void var_array2([in] long param[]);
void var_array3([in] long *param);
```

In a CORBA IDL file you may use the `sequence` keyword to define array types. The CORBA C Language Mapping defines that `sequence` is translated into a `struct`. Consider the following example:

```
IDL:
typedef sequence<long, 10> my_long_10;

C:
typedef struct
{
    unsigned long _maximum;
    unsigned long _length;
    long *_buffer;
} my_long_10;
```

You may have noticed that the boundary of the sequence is silently dropped when converting the sequence to the C type. You are responsible for enforcing the boundary yourself¹.

The syntax of CORBA allows you to specify a sequence as type of a parameter. When mapping this to C, this will produce a struct definition inside a function declaration. This implies that the struct type is only known inside the function, which means you cannot declare a variable of this type outside the function.

Strings

Strings are regarded as a variable sized array of characters, which is zero-terminated. This allows to omit the specification of length attributes for the parameter. Instead you specify an `string` attribute. The generated code will use string functions to determine the length of the string.

```
void hello([in, string] char* world);
```

In CORBA IDL you may specify strings with boundary (similar to the sequence type). The CORBA C Language Mapping defines to map strings and wide-strings to `char*` and `wchar*` respectively, ignoring the boundaries. This implies that you may have to enforce the boundaries yourself.

¹If you think this is unfair, please contact the OMG.

```
IDL:
typedef string<25> my_name;
```

```
C:
typedef char* my_name;
```

1.3.3 Structured Types

Within the IDL file you may specify a structured type—a **struct**—the same way as you would within a C/C++ header file. Furthermore it is possible to specify attributes with each member of the defined **struct**. One of the allowed attributes is **size_is**, which is described in more detail in Section 1.6. It allows you to specify the run-time size of an array. The definition of such a structure could look like this:

```
struct _string
{
    [size_is(length), string] char* buffer;
    unsigned length;
};
```

This is similar to the usage as parameters:

```
[in, size_is(length), string] char* buffer,
[in] unsigned length
```

A **struct** is mostly marshaled by copying it as is into the message buffer. If the **struct** contains variable sized members, such as the **buffer** member of the above example, these are marshaled separately after the parameter of the structured type.

Bit-fields

You may specify bit-fields in a structured type. Consider that this is highly platform dependent and the usage of bit-fields may lead to unwanted results.

```
struct {
    unsigned int first : 3;
    unsigned int second : 5;
    unsigned int third : 8;
};
```

1.3.4 Unions

Within your IDL specification you may define a union in C syntax or in IDL syntax. The difference is made when marshaling the union into the message buffer. A “C-style” union is simply copied into the message buffer. Therefore it will consume the space of the largest of its members.

The “IDL-style” union declares a decision making variable, which is then used to select the member to transmit. Such a union could look like this:

```
union switch(long which_member) _union
{
case 1:
    long l_mem;
case 2:
    double d_mem;
case 3:
case 4:
    long array[100];
default:
    byte status;
};
```

If the variable `which_member` has the value 1, the member `l_mem` is transmitted. When defining a switch variable and the case statement, you have to consider that they are used in a C switch statement for comparison. Thus it is valid to use a `char s_var` and define the cases with character values.

The generated code for an “IDL-style” union includes a switch statement to decide which member to transmit. This may be relevant if the union may tremendously vary in size and a comparison would possibly minimize the amount of data to be copied. But, such a union will always need the switch variable to be transmitted as well — at the receiver’s side there is also a switch statement which decides what to unmarshal from the message buffer.

A union may also have constructed or variable sized members. Members of variable size are marshaled after the union.

1.4 Deriving Interfaces

You can derive an interface from another interface. This is done using similar syntax as with C++ classes. You name the interface to derive followed by a colon and then the list of base interfaces.

```
interface derived : base1, base2
{
...
}
```

The main difference to C++ class derivation is, that you mostly use derived classes to overload functionality of the base classes. Since there is no implementation in an IDL file, which could be overloaded, the main purpose of interface derivation is the expansion of the base interface' functionality. This is done by specifying new functions in the derived interface.

You may use interface derivation to unify multiple interface definitions into one interface. The derived interface may be empty.

```
interface all_in_one :
    base1, base2,
    another_scope::base3
{};
```

The generated server loop will be able to receive message from `base1`, `base2`, and `another_scope::base3`. The server loop distinguishes the different function calls from another as described in the following section (Section 2.5).

1.5 Constants

You may specify constant by either using the `#define` statement or a `const` declaration. The `#define` statement is parsed by the pre-processor and will not appear in the generated code. The `const` declaration will appear in the generated code as they have been defined in the IDL file.

```
const int foo = 4;
```

You may use the declared `const` variables when specifying the sizes of arrays, etc.

1.6 Attributes

This section describes some attributes, which are available with the DCE IDL. If an attribute is available for CORBA IDL as well this is specifically mentioned.

Attributes are used to propagate knowledge about the target environment or usage context of the IDL specification to DICE. Some attributes

are used to generate code optimized for a specific platform or architecture. Others will influence the generation of the target code to exploit features of a specific L4 ABI. Therefore the attributes are important to provide DICE with knowledge about the IDL specification and optimization potential.

However, there are attributes which might be ignored on some platforms.

1.6.1 Array Attributes

Attributes describing parameters of an array mostly end on `_is` and usually take a parameter. They are: `first_is`, `last_is`, `length_is`, `min_is`, `max_is`, and `size_is`. Currently only `length_is`, `size_is`, and `max_is` are supported.

Starting with the latter, the attribute is used to determine the maximum size of an array. For example do

```
[in] int parameter[100]
```

and

```
[in, max_is(100)] int parameter[]
```

express the same semantics.

You may also specify a parameter or constant as the argument of the attribute.

The `length_is` or `size_is` attributes are used to determine the length of a variable sized array at run-time. The generated code will transmit only the specified number of elements instead of the maximum size. The `str` parameter does not have a maximum size to keep the example simple. A maximum size will enable the server loop or client stubs to know the size of memory to allocate for the array.

The example:

```
[in, size_is(len)] char str[],
[in] int len
```

specifies a variable sized array of bytes, which has a length of `len`. `len` has to be either a parameter or a constant. DICE first searches for parameters with the specified name and, if it does not find one, it searches for a constant with that name. An error is generated if neither is given.

1.6.2 Strings

As described above is it possible to define the run-time length of a variable sized array using the `length_is` or `size_is` attribute. This may also be used for strings. If you know that the parameter you will transmit is a zero-terminated string, you can also let the generated code calculate the size of the string. You simply specify the `string` attribute.

```
[in, string] char* str
```

In this example the `str` parameter does not have a maximum size, because it can be any size.

1.6.3 Indirect Parts IPC

L4 allows you to transmit any data using indirect part IPC. You specify the address and size of the original data and at receiver's side the address and size of the receive buffer and the kernel copies the data directly. Without indirect parts the data is copied from the source address to the message buffer, the kernel copies the message buffer from senders side to the receiver's message buffer, and the receiver copies the data from the message buffer to the target variables. The downside of the indirect part IPC is, that the kernel may have to establish additional temporary mappings of the data areas.

To make DICE use indirect part IPC for a specific parameter you may use the `ref` attribute. You have to combine it with either the `size_is` attribute or the `string` attribute to specify the size of a variable sized array.

```
[in, size_is(size), ref] long data[],
[in] long size
```

You may also transmit any other kind of data using indirect part IPC. Only specify the `ref` attribute with the respective parameter and the size of the data to be transmitted.

With the L4 version 4 back-end all constructed data types will be transmitted using indirect parts, even if no `ref` attribute has been specified (or rather, the compiler may choose parameters to transmit as indirect parts as it seems fit).

Be aware that for indirect parts with the `[out]` attribute the receive buffer has to be allocated before the IPC is initiated. Thus, the `size` parameter also has to have an `[in]` attribute. The `[out]` attribute has to be present, so that the `size` parameter is used to set the send size of the indirect part.

1.6.4 Receiving Indirect Parts

The CORBA C Language Mapping specifies for the reception of variable sized parameters the allocation of memory using the `CORBA_alloc` function. If you know the receive buffer for the variable sized parameter before calling the client stub, you may want to give this buffer to the client stub, so the data is copied directly into this buffer. This can be done using `prealloc_client` attribute:

```
[out, prealloc_client, size_is(len), ref] char** str,
[out, in] int *len
```

The Section 2.8 contains further information on the memory management of indirect parts.

Even though, the receive buffer for the indirect part has been allocated beforehand, the size element of the indirect part is set to the value of the `len` parameter. Thus, this parameter should have the `[in]` attribute to indicate this situation. Also, the very parameter is used to set the actual transmitted size when sending from the server to the client. Therefore, you should set this variable to the actual size of the buffer to be sent back to the client. Otherwise the kernel might abort the IPC with an error (if the size specified is larger than the buffer).

Also note that the `str` parameter has an additional reference as `out` parameter even though it is preallocated and thus no memory allocation is necessary in the stub.

Note: The `prealloc_client` attribute also has the counter-part `prealloc_server`. Using this attribute will generate code that allocates memory for the parameter in the server side dispatcher function. The allocated memory is then handed to the component function for using. Because the environment's `malloc` (or the `CORBA_alloc`) function is used for this parameter on every invocation of the respective operation, the memory should eventually be freed. This is automatically done by the generated code. It registers the allocated memory pointer in the environment's pointer list. This list is iterated after the receipt² of the next message and freed. If you want to keep the memory for further processing, you have to remove the memory from the list. Usually the usage of `prealloc_server` is not necessary.

²Freeing the memory before sending the message is not possible, because the memory is needed during the reply.

1.6.5 Error Function

Usually the server ignores IPC errors when sending a reply to a client or waiting for a new request. Sometimes it is useful to use timeouts when waiting for new request to “do something else” meanwhile. The wait timeout can be specified in the `CORBA_Environment` parameter of the server loop. But how do you know when the timeout has been triggered and what does the server loop do?

For this scenario, there exists an attribute — `error_function` — which makes the server loop call the specified function on every IPC error. This allows you to implement a specific behavior for the IPC error. The called function takes only one parameter — an `l4_msgdope_t` variable. This can be used to check the origin of the error and take appropriate measures. To distinguish client and server error functions, one can use the `error_function_client` and `error_function_server` attributes respectively. Using these attributes overrides `error_function`.

1.6.6 Using other types for transmission

Sometimes it is convenient to use other types to transmit the data, than are used when calling a function. An example is the usage of a constructed type, which contains variable sized members, which are known only at run-time. If this type is declared in a C header file, it is not possible to add attributes to the variable sized member to define the run-time size.

The attribute `transmit_as` lets you define the type to use when transmitting the data:

```
typedef [transmit_as(unsigned long)]
my_C_union idl_union;
```

This example will use the type `unsigned long` instead of `my_C_union` when transmitting parameters of type `idl_union`.

The example:

```
[in, transmit_as(unsigned char),
 size_is(size)] var_struct *t,
[in] unsigned size
```

will transmit `t` as a variable sized array of type `unsigned char` with `size` instead of `var_struct`.

Consider that the latter example requires the specification of a pointer parameter. Without the `transmit_as` attribute this would have the semantic of

an array of type `var_struct` and `size` members. But with the `transmit_as` attribute the parameter `t` is considered a variable sized array with `size` elements of type `unsigned char`.

1.7 Imported and Included Files

DICE provides you with two mechanisms to include other IDL or C/C++ files into your IDL file. The first is the well known preprocessor directive `#include`. The second is the `import` statement. Both are resolved by DICE's preprocessor. Including a file using `import` implies that the content of this file is only scanned for IDL syntax. This includes C/C++ style type and constant definitions, but excludes C style function declarations and definitions.

DICE automatically scans the file `dice-corba-types.h` to know all the CORBA types. This file is searched in L4 tree's include path and the include path where dice headers might be installed to. To avoid inclusion of unwanted header files the two include paths contain the `/dice` sub-directory. If you receive the error '`dice-corba-types.h` cannot be found', please check if your `$(L4DIR)/include/dice` or `/usr/local/include/dice` contain that file. If not, fix your installation or run `make` in the DICE source directory.

1.7.1 C/C++ Attributes

If you specify attributes (`__attribute__((...))`) with a type definition, they are ignored by the IDL compiler. This is important if you specify attributes responsible for the memory layout of a data type. Thus, if you define C types using attributes check the generated C code to see if you data type is really copied the way you intended. (For example, the `packed` attribute of structs.)

1.8 Message Passing

Traditional RPC semantics include the sequence of actions of sending a request to a server, processing the request and waiting for a reply. Opposed to that, exist programming languages, which use communication mechanisms to synchronize or exchange data at specific points in the program. In Ada these points are called "rendezvous". With this technique it is possible to write programs, where activity is associated the flow of data.

To provide the level of abstraction we introduced for RPCs for message passing also, we extended the DCE IDL to let DICE generate simple communication stubs.

A “rendezvous” point is given on L4 when one thread waits for a specific message and another thread is sending this message to the waiting thread. Since a message is defined in the IDL by an operation of an interface, we associate an attribute with the operation to denote the message passing semantic. It is possible for a message to be consumed by a thread or emitted by it. This is similar to the semantic for parameters of a RPC. Thus we used the `in` and `out` attribute for message passing.

```
interface mp
{
  [in] void send(int param);
  [out] int receive(int *param);
};
```

The parameters of an `in` operation are automatically `in` as well. The usage of return types other than `void` or `out` parameters will cause an error. Accordingly are all parameters of an `out` operation `out` themselves.

DICE generates a different set of functions for message passing operations. At the sender’s side³ the functions end on `_send`, because they only involve the send phase of the IPC. At the receiver’s side there are two functions, one ending on `_wait` and the other ending on `_recv`. These functions receive the specified message from any sender or a specific sender respectively. For the receiver’s side there also exists an `_unmarshal` function.

If the receiver’s side is also the server’s side the message can also be received by a server loop. For this case DICE also generates the before-mentioned functions (`_unmarshal` and `_component`).

A synonym for a function’s `in` attribute is the `oneway` attribute, known from DCE IDL.

1.8.1 Optimizations

If you would like to use message passing stubs and know what you are doing, you might want to use some extra attributes. Assuming you do use the message passing function as `send` and `receive` pairs only—meaning no single spot where you wait for more than one message. You can specify the `noopcode` attribute for a function. This attribute makes DICE skip opcode

³Beware, this can be the client’s side *and* the server’s side.

marshaling and unmarshaling. This save one message word for some kernel interfaces (V2 and X0). But it bares the generated code to validate if the message you received is really the message you wanted to get (which is otherwise done using the opcode).

A similar function attribute is the `noexceptions` attribute. When using simple RPC mechanisms, e.g. function calls, an exception is usually returned. This allows the caller to check, for instance, if the server recognized the opcode. Using the `noexceptions` attribute generates function stubs, which do not transfer this exception code back from the server to the calling client. This provides an additional message word for data transfer.

1.9 Asynchronous Servers

Asynchronous servers are servers, which receive a request, enqueue or propagate it and immediately wait for the next request. At a later moment (asynchronously) the reply is delivered to the client. To enable asynchronous processing of a function, use the attribute `allow_reply_only` for this function. The `*_component` function will have an additional parameter (`short *_dice_reply`) which is set to the value of `DICE_REPLY`. This implies that the component function does work synchronously per default. To delay the reply set the value of `_dice_reply` to `DICE_NO_REPLY` before returning.

When the work is finished you can sent a reply to this request to the client using the `_reply` function generated for this function. It expects the `CORBA_Object` for the client, the returned parameters, and a reference to a `CORBA_Server_Environment`.

Example

The following asynchronous server has one function, which receives a parameters and returns a result. This function should be processed asynchronously:

```
[allow_reply_only]
void foo([in] long p1, [out] long* p2);
```

To return the result to the client, the server has to respond from the same thread as the client sent the request to. Therefore we will need a function, which does handle the replies:

```
void notify([in] long p1, [in] long p2);
```

`p2` is not a reference, because it is a simple `in` parameter for the `notify` function. `p1` is used as an identifier for the job – it has to be unique.

The `_component` function may create a worker thread, which does all the processing of the job. To be able to send the result of the job back to the client the server will need to store the association of client ID to job ID. (In fact the client ID could be used as job ID if a client ID cannot be used again while the job is still being processed, e.g. the client did not receive a reply yet.) The worker thread should be informed about the job to process.

After the job is finished the worker thread should invoke the `notify` function with the job ID (`p1`) and the result (`p2`). The implementation (`_component` function) of this method calls the `_reply` function of the `foo` method.

Remarks

To send a reply to a client you have to store its ID and an identifier for the job/request. (Remember: `CORBA_Object` is a pointer.)

For some architectures (e.g. L4 version 2) the client initiated a call which expects the reply from the same thread as the request has been sent to. Therefore, you will need for each asynchronous function a function in the server loop which delivers the replies.

Inspect the example `async` for an implementation example.

1.10 Thoughts About DICE Specific Types

When using IDL types, they mostly can be mapped to L4 specific message types using attributes. An indirect part is for instance denoted by the `DCE ref` attribute. It thereby follows the semantic declared in the DCE IDL specification.

However, there are some L4 specific IPC semantics, which are hard to express using attributes. The following few sections will try to explain the semantic of some of the L4 specific types and ideas on how to represent them using other methods than currently used ones (`flexpage` type).

1.10.1 flexpage

A flexpage is a memory region consisting of at least one memory frame which can be transferred from one address space to another (or rather the rights to access it). It consists of a base address, a size, and the transferred rights, including the ownership. There also apply restrictions to the address — it

has to be page size aligned — and the size of a flexpage. To express this using attributes, we could write something like:

```
[in, memory, size_is(pages),
rights(access)] void* base_address,
[in] unsigned int pages,
[in] unsigned int access
```

This scheme would add two new attributes (`memory` and `rights`) and require the user to know all necessary parameters. This can be simplified by using a predefined type, such as:

```
typedef struct {
[memory, size_is(pages), rights(access)]
void *base_address;
unsigned int pages;
unsigned int access;
} flexpage;
```

An IDL compiler should map this constructed type onto the existing `l4_fpage_t` type of L4 and use it appropriately. Since one of the goals of the IDL compiler is the usage of existing data types — such as `l4_fpage_t` — this would be contra productive.

Therefore we decided to use the special type `flexpage` or `fpage`. It is directly mapped to the `l4_fpage_t` type and implies the above mentioned characteristics of a memory region. It is simpler in the usage, because the user does not have to transfer the L4 type to the predefined IDL type.

1.10.2 Interrupts

One of the future goals is to make the interaction with the kernel transparent to the user as well. One could imagine to describe the kernel interface using an IDL file, which defines all type and system calls and the IDL compiler generates the appropriate code to invoke the system calls.

This idea involves the abstraction of interrupt IPC. Therefore an IDL type for interrupts should exist, which can be used to describe an interrupt IPC.

These are abstract ideas, which have not been implemented and will not be in the near future.

Chapter 2

Using the Generated Code

To illustrate the usage of the generate code, the following example is used. To look up the syntax of the example consult the previous section. This example IDL is contained in a file named `test.idl`.

```
library example
{
    interface first
    {
        void foo([in] int parameter);
    };
};
```

2.1 Invoking DICE

DICE generates per default code for both client and server side. To translate the above example into target code for the C language and the L4 version 2 ABI on a x86 architecture, call:

```
dice test.idl
```

The target language C, as well as the L4 version 2 ABI and the x86 architecture are default setting for the respective options. The above call will generate five files.

1. `test-client.h`
2. `test-client.c`

3. `test-server.h`
4. `test-server.c`
5. `test-sys.h`

The file `test-sys.h` contains the opcodes for the interface as described in Section 2.5. The client and server side code is contained in the respective files.

2.2 Client Code

The client side code packs the in data into a message buffer that is send via IPC to the server. The declaration of the client function looks like this:

```
void example_first_foo_call(CORBA_Object _dice_corba_obj,
    int parameter,
    CORBA_Environment *_dice_corba_env);
```

There are several things noteworthy. Two additional parameters have been added to the function. According to the CORBA C Language Mapping [1]. The first identifies the server, which is on L4 a pointer to the thread ID (`l4_threadid_t`) of the server. `CORBA_Object` is a type alias for a pointer to an L4 thread ID. The last parameter is the CORBA environment, which is a collection of information denoting the context of the call. For more detailed information about the `CORBA_Environment` see Section 2.6.

If using C as target language, the name of the function is composed of the name of the library or module (`example`), the name of the interface (`first`), and the name of the function (`foo`).

2.3 Server Code

The function template generated for the server looks similar to the client side code. The only difference is the function's name. The parameter list is the same as the client side function.

```
void example_first_foo_component(CORBA_Object _dice_corba_obj,
    int parameter,
    CORBA_Environment *_dice_corba_env);
```

At the server side the `CORBA_Object` parameter does not contain the ID of the server, but instead the ID of the sender. It can be used to identify the sender of a request.

By default the server function is only declared in the server header file, but no implementation exists. One of the options `-t` or `--template` will add an additional file which ends on `-template.c` to the list of generated files. It contains function skeletons for the server functions. Copy these skeletons and add the functionality of the server. Note that every new invocation of DICE with the mentioned options will regenerate these templates.

2.4 Server Loop

The server's side of the communication implements functionality, which receives messages, dispatches them to the appropriate server side functions, and sends replies to the client. This process can be separated into the following steps:

1. receive any message
2. get opcode from message
3. check opcode, and depending on it
4. unmarshal the `in` parameters
5. call the server side function
6. marshal the `out` parameters
7. send the reply
8. start from the beginning (step 1)

For performance reasons the reply and wait for new messages is usually combined into one step. Accordingly to the above steps, DICE generates helper functions which have the following endings:

1. and 2. `<interface>_wait_any`
3. is a switch statement inside of `<interface>_dispatch`
4. `<function>_unmarshal`
5. `<function>_component`

6. `<function>_marshal`

7. and 8. `<interface>_reply_and_wait`

The functions to `<interfac>_wait_any`, `<interface>_dispatch`, and `<interface>_reply_and_wait` are called inside of the `<interface>_server_loop` function. The `<function>_*` functions are called in the `<interface>_dispatch` function.

The place holders `<interface>` and `<function>` show that the name is either interface or function specific. The aforementioned example produces an `<interface>`-specific function with the name `example_first_wait_any` and a `<function>`-specific function with the name `example_first_foo_unmarshal`.

The `<interface>_server_loop` function does take a `void *` parameter. It does not return any value.

For an interface which is derived from other interfaces, the server loop will also be able to distinguish messages for functions of the base interface. It dispatches these messages to the appropriate server functions.

2.5 Function Identifiers — Opcodes

To distinguish the messages from one another, e.g. to know which function should be called, the message contains at a defined position a number, consisting of an interface identifier and a function identifier — the *opcode*. This opcode is a number, which is determined by the sequence of the declaration of the functions inside the interface definition. For example does DICE generate for the following IDL:

```
interface simple
{
    void func1();
    void func2();
    void func3();
};
```

the function identifiers:

1. `func1`: 1,
2. `func2`: 2, and

3. `func3`: 3.

If you derive an interface from `simple`, the function identifiers would have to continue starting with 4. To avoid this, the interface identifier is used. For example do the functions of the following interface

```
interface derived : simple
{
    void func4();
    void func5();
};
```

map to the following function identifiers:

1. `func4`: 1 and
2. `func5`: 2.

The functions `func1` and `func4` are only different in the interface identifier, which is 1 for `simple` and 2 for `derived`. The resulting opcode is generated by shifting the interface identifier left by `DICE_IID_BITS`¹ and then AND the function identifier bitwise. Thus the opcode for `func1` is `0x100001` and for `func4` is `0x200001`.

2.5.1 Defining Opcodes

To determine the interface and function identifiers yourself, you may use the `uuid`² attribute for the interface or functions. If used with the interface, for example:

```
[uuid(0xC00)]
interface other
{
    ...
```

it will only effect the interface identifier. Valid values for interface identifiers range from 1 to `0xFFF`. If you specify an interface identifier, which is already used by a base interface, the function identifiers of the derived interface will be counted starting after the biggest function identifier of the base interface. For example does:

¹The value is currently set to 20 and is defined in one of the dice header files.

²`uuid` stands for Universal Unique Identifier and is supposed to contain a 128 bit number. The DCE Specification allows the `uuid` attribute only for interfaces. We (mis)use it for interface and function identifiers.

```
[uuid(1)]
interface derived : simple
{
...

```

generate the function identifiers:

1. func4: 4 and
2. func5: 5.

You may use the `uuid` attribute with functions as well. Valid values are within the range of 0 to `0xF'FFFF`. Function identifiers are first assigned using the `uuid` attribute. Then the remaining functions are numbered starting from the lowest, not-assigned number. For example:

```
interface simple
{
    void func1();
    void func2();
    [uuid(1)] func3();
};

```

will generate:

1. func1: 2,
2. func2: 3, and
3. func3: 1.

If a derived interface has the same identifier as the base interface it will start to enumerate its own functions with the highest number of the base interfaces. The example:

```
interface simple
{
    void func1();
    [uuid(4)] void func2();
    [uuid(1)] void func3();
};

```

```
[uuid(1)]
interface derived : simple
{
    void func4();
    void func5();
};
```

will generate the following identifiers:

1. `func1`: 2,
2. `func2`: 4,
3. `func3`: 1,
4. `func4`: 5, and
5. `func5`: 6.

This implies that any changes to the IDL of a base interface imply re-generation of code generated for derived interfaces.

It is possible to use constants instead of numbers with the `uuid` attribute.

2.6 CORBA Environment

The CORBA Environment is split into two types. One environment for the client side and one for the server side. This is done because client wrapper functions often declare the environment on the stack. To keep the memory footprint small, the client side environment is a reduced version of the server side environment.

The respective definitions can be found in [Appendix D](#).

The first elements of the `CORBA_Environment` are defined by the CORBA C language mapping. The first is of type `CORBA_exception_type` (an `int`), which consists of a `major` exception number, and a `repos_id`. The major number defines whether the exception was a system exception (e.g. communication error) or a user defined exception. The second member defines a minor exception number.

The `repos_id` can also be used as an index into a repository with descriptions of the exceptions. It can be used to print precise error messages. The exception description can be accessed using the method `CORBA_exception_id`, which takes the `repos_id` as an argument.

The third parameter is also defined by the CORBA C language mapping. It is a void pointer to user defined data. User defined means here, that it is defined by the exception setting instance. An exception can be set using the method `CORBA_exception_set`. Functions to manipulate the CORBA Environment are explained in the following sections.

This parameter is used disjunct with the IPC error code. If there was an IPC error, no exception could be transmitted.

2.6.1 CORBA Environment Functions

There are a few functions defined by the CORBA C Language mapping [1] to manipulate the CORBA Environment.

```
void CORBA_exception_set(
    CORBA_Environment *ev,
    CORBA_exception_type major,
    CORBA_char *except_repos_id,
    void *param);
```

The above function is used to set the exception part of the environment. It receives a pointer to an environment structure to be initialized. It first calls `CORBA_exception_free` for the environment. Then it sets the `major` member of the environment and if it is not equal to `CORBA_NO_EXCEPTION` the `repos_id` and `param` members are set as well.

```
CORBA_char* CORBA_exception_id(CORBA_Environment *ev);
```

This function retrieves the stored exception description for the `repos_id` of the environment. The exception descriptions are stored in the global variable `__CORBA_Exception_Repository` and are currently set as described in table 2.1.

Table 2.1: Stored exception descriptions.

repos_id	stored string
<code>CORBA_DICE_EXCEPTION_NONE</code>	none
<code>CORBA_DICE_EXCEPTION_WRONG_OPCODE</code>	wrong opcode

```
void* CORBA_exception_value(CORBA_Environment *ev);
```

This function returns the value of the `param` member of the environment.

```
void CORBA_exception_free(CORBA_Environment *ev);
```

This functions is supposed to free any allocated memory of the resource. Since `CORBA_exception_set` does not allocate any memory, the members are simply set to the default values: `major` is set to `CORBA_NO_EXCEPTION`, `repos_id` is set to `CORBA_DICE_EXCEPTION_NONE`, and `param` is set to zero.

2.6.2 DICE Specific Extensions

DICE currently adds some members to the environment. They are `timeout`, `rcv_fpage`, `ipc_error`, `user_data`, `malloc`, `free` and memory pointers `ptrs`.

The `timeout` value is used to set the timeout of an IPC. It is of type `l4_timeout_t`.

The `rcv_fpage` value is used to set the receive window for received flex-pages. It is of type `l4_fpage_t`.

The `ipc_error` value is set if an IPC error occurred to the value of `L4_IPCERROR(result)`, where `result` is the result variable of the IPC³. If there was no IPC error the value is undefined. To determine if there was an IPC error test the `major` member of the environment for `CORBA_SYSTEM_EXCEPTION`.

The `user_data` member is of type `void *` and can be used freely by the user. It is ignored by the generated stubs and not transferred from the client to the server, but consistent within the scope of the server loop or client stubs.

The `malloc` and `free` members are a function pointers to memory allocation routines of type

```
void* (*malloc) (unsigned long);
void (free) (void*);
```

which are used to allocate and free memory. These function pointers are currently used by default.

When using an `CORBA_Environment` you should always initialize it with `dice_default_environment`. This ensures, that all members are set to default values. Regard that the memory function pointers are set to default functions, which enter the kernel debugger. The default environment of `CORBA_Server_Environment` is `dice_default_server_environment`.

³An L4 IPC returns a result variable, which contains information about the transferred number of double words and indirect parts. If there occurred an error during the IPC, the result variable contains further information about the kind of the error.

If you assign the memory function members of the environment new functions, some compilers complain about mismatching types. Simply cast your function using `dice_malloc_func` and `dice_free_func` respectively.

2.7 Buffer Management

This section explains the different kinds to manage memory buffers with DICE.

2.8 Memory Management for Indirect Parts

Since the server loop is auto generated, it has to use heuristics to provide the receive buffer for indirect parts. The default approach is to dynamically allocate the memory using the `malloc` member of the `CORBA_Environment` of the server loop. If this member is not set it uses the `CORBA_alloc` function. You can force the usage of `CORBA_alloc` by using the option `--fforce-corba-alloc` (see Section 3.1.4). Another alternative is the use of the `init_rcvstring` attribute (see below for explanation).

2.8.1 malloc and CORBA's alloc

The `malloc` member of `CORBA_Environment` and the `CORBA_alloc` functions have the same syntax as the libc `malloc` function. They take the desired size of the memory as parameter and return a `void *` to the allocated memory.

Example for `malloc` member of `CORBA_Environment` (the IDL file has to be translated using `-fserver-parameter`):

```
CORBA_Environment env = dice_default_environment;
env.malloc = (dice_malloc_func)malloc;
env.free = (dice_free_func)free;
server_loop(&env);
```

Example for `CORBA_alloc`:

```
void* CORBA_alloc(unsigned long size)
{
    return malloc(size);
}
```

The server loop will call the `malloc` or `CORBA_alloc` function for every receive buffer it should set. It defines the size of the buffer by the maximum of all receivable indirect parts. To determine the size of an indirect

part, DICE uses the maximum values defined either in brackets or using the `max_is` attribute. If the parameter is still considered variable sized, a heuristic is used to determine the maximum size: all variable sized strings are “defined” to have a maximum size of 512 bytes. All other variable sized arrays are defined to have a maximum size of 1024 bytes. If you want to send data larger than these sizes use the `max_is` attribute to give DICE a hint for the receive buffer.

2.8.2 `init_rcvstring`

Example for `init_rcvstring`: first the IDL file, then the needed additional code:

```
[init_rcvstring(my_alloc)]
interface simple
{
    void foo([in, ref] long data[200]);
};
```

DICE generates a server loop, which initializes each receive buffer for indirect parts by calling the named (`my_alloc`) function. A possible implementation for this function might look like this:

```
void
my_alloc(int nb,
         l4_umword_t *addr,
         l4_umword_t *size,
         CORBA_Environment *env)
{
    *addr = my_buffers[nb];
    *size = BUFFER_SIZE;
}
```

It receives four parameters:

1. a zero based index indicating which of the strings should be set (`nb`),
2. a pointer to the variable which should hold the address of the buffer (`addr`),
3. a pointer to a variable which should hold the size of the buffer (`size`), and
4. a pointer to the current `CORBA_Environment`.

Chapter 3

A Short Reference to Compiler Options

This chapter will provide you with a short overview about the available compiler options. It will also show you what you can do with these options.

Most of the information given here can also be found in the manual page available for DICE.

3.1 Compiler Options

There are several compiler options which are roughly grouped into general compiler options, preprocessor options, and back-end options.

3.1.1 Pre-Processing Options (Front-End Options)

These options affect the preprocessing of the input files.

`-x <language>`

Specify the language of the input IDL files. Possible values are `dce` and `corba`. This option defaults to `dce`. Include header files are recognized by their extension.

`--preprocess, -P`

Passes the arguments given to this option to the preprocessor.

Example: `-P--nostdinc`

`-Wp,`

Same as `-P`. `-P` and `-Wp`, are scanned for `-I` and `-D`.

`-I`

Provides the preprocessor with include directories to search for the files specified with the `#include` and `import` directives. (Same as `-P-I`.)

Example: `-I/usr/include`

`-nostdinc`

Passed as is to the preprocessor.

`-D`

Provides the preprocessor with additional symbols. (Same as `-P-D`.)

Example: `-DL4API_14v2`

`-E`

Stop “compilation” after preprocessing. The output of the preprocessor is printed to `stdout`. Consider that the preprocessor does not resolve `import` statements and therefore these statements will appear in the generated output.

`-EXML`

Stop “compilation” after the parsing. The intermediate representation (the abstract syntax tree – if you like) is printed into an XML file. The name of the XML file is the same as the IDL file’s with the `idl` extension exchanged against the `xml` extension.

`-M`

Print include file tree and stop after doing that.

`-MM`

Print include file tree for files included with `”` and stop after doing that.

`-MD`

Print include file tree into `.d` files and compile.

`-MMD`

Print include file tree for file included with `"` and compile.

`-MF <filename>`

Generates the dependency tree into the file specified by the filename. This option only works if one of `-M`, `-MM`, `-MD`, or `-MMD` is given.

`-MP`

Generates for all files, which the generated files depend on, a phony dependency target in the dependency list. This option also requires one of the dependency generation options.

`--with-cpp=<argument>`

Specify your own preprocessor. This will override environment variable `CC` or `CXX`.

Example: `--with-cpp=/usr/bin/cpp-3.0`

3.1.2 Back-End Options

`--client, -c`

Create client side code only. Default is to create both client and server side code.

`--server, -s`

Create server side code only. Default is to create both client and server side code.

`--template, -t`

Create server skeleton/template file.

`--no-opcodes, -n`

Do not generate the opcode file. This is useful if you wish to specify the opcodes yourself or with the parameter `-fopcodesize`.

`--filename-prefix, -F`

Prefix each file-name of the DICE generated files with the given string.

Example: `-FRun1` lets the includes of the generated files look like this:
`#include "Run1<file>-client.h"`.

`--include-prefix, -p`

Prefix each file-name inside the generated include statements with the given string, which is interpreted as path.

Example: `-p/tmp/dice` lets all includes be prefixed with the string:
`#include "/tmp/dice/<file>-client.h"`.

Together with `-F` the generated include statement looks like this
`#include "/tmp/dice/Run1<file>-client.h"`.

`-o`

Specify output directory. All generated files are placed into the given directory. If an invalid directory is given you receive error messages stating that files cannot be opened.

`--create-inline=<mode>, -i<mode>`

Generate client stubs as inline. You may specify a mode for inline. Use `static` or `extern`. The mode is optional. This will only generate header files, since the implementations will appear there.

Example: `-istatic` generates `static inline` functions.

`-B<argument>`

Defines the back-end which is used to generate the target code. There are three categories, which define a back end. The first is the *target platform*, which is denoted by the `p` suffix to the `-B` option. The second is the *target kernel interface*, denoted by `i`. And last is the *target language mapping*, denoted by `m`.

The target platform can be one of `ia32`, `arm`, or `amd64`. If none of the mentioned platforms is chosen, `ia32` is used.

The `arm` platform is currently only supported with the X0 kernel interface.

The `amd64` platform is currently only supported with the V2 kernel interface.

Example: `-Bpia32`

The target kernel interface can be one of `v2`, `x0`, `x0adapt`, `v4`, `sock`, or `cdr`. Currently `v4` is not fully supported. The kernel interface `x0adapt` uses the X0 kernel interface, but presents a V2 interface to the user. The difference between the kernel interfaces is in the size of the `l4_threadid_t` type. Using `x0adapt` you can continue to use your applications written for use with a V2 `l4_threadid_t` type on top of an X0 kernel. The `x0` kernel interface is badly tested, because there currently exists no L4Env for native X0.

Example: `-Bix0adapt`

The target language mapping can be one of `C` or `CPP` – for C++. Using the C++ back-end will generate code that can be compiled with `g++`. It does not contain classes for interfaces yet.

Example: `-BmC`

`-O<level>`

This option sets the optimization level of the back-end. *deprecated! do not use!*

`--testsuite, -T`

Generates a test-suite for the declared interfaces. The test-suite will start an L4 thread with the server, and call all functions with random values for the parameters. The `in` parameter values are checked at the server for conformance with the values set at the client side. The `out` parameter values are set in the server implementation function and checked at the client's side.

This option will generate the `-template.c` file with implementations of the server function which test the values. It will also generate a `-testsuite.c` file which initializes the parameters and checks `out` parameters. It also contains code to start the server thread and initialize the server loop.

If you wish to write your own test-suite, don't use this option, because this option makes DICE generate target code which is complete and won't use your test functions.

`--message-passing, -m`

Generate message passing functions for RPC functions as well. See Section 1.8 for more details.

3.1.3 General Options

This section describes some of the options, which do not fit into one of the above categories.

`--help`

Displays a verbose help screen, showing all of the available options.

`--version`

Displays version information, including the build date and the user who built this version.

`--verbose, -v<level>`

Displays verbose output. The optional value specifies the amount of verbosity. The higher values the more output.

`-f`

Specifies additional compiler flags, which are hints for the compiler on how to generate code. See section 3.1.4 for details.

3.1.4 Compiler Flags

The mentioned flags are specified as argument to the `-f` option. So the argument `ctypes` is used as `-fctypes`.

F - Filetype

This is a “nested” option, which itself takes arguments. These arguments are described in table 3.1.

Argument	Alternative	Meaning
idlfile	1	Generate one client implementation file per input IDL file.
module	2	Generate one client implementation file per specified module.
interface	3	Generate one client implementation file per specified interface.
function	4	Generate one client implementation file per specified function.
all	5	Generate one client implementation file for all IDL files.

Table 3.1: Filetype Options

ctypes

This option specifies that the generated code should use C types rather than CORBA's C types. `long` is used instead of `CORBA_long`. CORBA types, which have no C expression, are used as CORBA types.

This option is respected in the Name Factory when generating type names.

l4types

This option specifies that the generated code should use L4 types¹ rather than CORBA C types. `l4_int32_t` is used instead of `CORBA_long`.

This implies that the generated code is used within an L4 environment, which knows these types. (L4Env does.)

This option is respected in the Name Factory when generating type names.

opcode size=<size>

This is a nested option. It can be used to determine the size used by the opcode within the message. Its possible values are shown in table 3.2.

¹L4 Types are type aliases for commonly used C types, which might have different size across platforms. The long type is sometimes 16, 32, or 64 bits wide. These types are defined in the `14/sys/14int.h` header.

Argument	Alternative	Meaning
byte	1	uses only 1 byte for the opcode
short	2	uses 2 bytes for the opcode
long	4	uses 4 bytes for the opcode (default)
longlong	8	uses 8 bytes for the opcode

Table 3.2: Opcode Size Options

DICE generated opcodes assume the opcode size of 4 bytes. If you specify other sizes than this one, you should also use the `--no-opcode` option, so you can specify appropriate opcodes.

`server-parameter`

If you specify the `-fserver-parameter` switch the server loop interprets the `void *` parameter as a pointer to a `CORBA_Environment` variable. It then uses the values of the environment to specify a receive window for flexpages, timeouts, etc. Without this option the parameter is ignored.

`no-server-loop`

If specified no server loop will be generated. Instead only the dispatch function will be generated.

`init-rcvstring=<function-name>`

Specifies a function to be used to initialize the receive buffers of indirect strings. This is the same as the `init-rcvstring` attribute you may specify with an interface. If you specify this option, the function is applied to all generated server loops, which have no `init_rcvstring` attribute.

`force-corba-alloc`

DICE uses by default the function `malloc` of the `CORBA_Environment` to dynamically allocate the memory for variable-sized receive parameters. To enforce the usage of the `CORBA_alloc` function, specify the option `-fforce-corba-alloc`. The usage of `CORBA_alloc` implies that there have to be potentially two implementations of `CORBA_alloc` — one for the client side and one for the server side. Also consider that an implementation

of `CORBA_alloc` in a client library may collide with a different implementation in another client library. If you use the `malloc` member of the `CORBA_Environment`, you can assign your implementation of `malloc` as you wish, e.g. `liba_malloc`.

You may detect the usage of `malloc` using the `-Wprealloc` option with DICE.

`force-c-bindings`

This option enforces the usage of the L4 API C bindings. Otherwise DICE may decide to generate inline assembler code for IPCs.

`trace-server=<function>`

If specified the generated server code contains tracing code, which prints status information to the LOG server.

If specified the `<function>` is used to print the trace messages. The function has to have the same signature as the `printf` function.

`trace-client=<function>`

If specified the generated client code contains tracing code, which prints status information to the LOG server.

If specified the `<function>` is used to print the trace messages. The function has to have the same signature as the `printf` function.

`trace-dump-msgbuf=<function>`

This option makes DICE generate code which dumps the content of the message buffer just before and after each IPC. *This may produce an immense amount of status output.*

If specified the `<function>` is used to print the trace messages. The function has to have the same signature as the `printf` function.

`trace-dump-msgbuf-dwords=<number>`

This option restricts the number of dumped dwords to `number`.

`trace-function=<function>`

Each of the `trace-*` options may be followed by `=<function>` to specify an output function for this class of traces. The specified function has to

follow the `printf` syntax, which means it has to take as first argument an format string and then a variable number of arguments.

To specify one function for all of the options, use `trace-function`. The default function is `printf`.

`zero-msgbuf`

This option lets DICE generate server code which zeros and then re-initializes the message buffer, just before marshaling the return parameter in the `wait` or `reply-and-wait` functions. This provides the marshaling code with a clean message buffer.

`use-symbols` or `use-defines`

DICE generates IPC code which contains a lot of `#ifdef` directives for profiling, frame-buffer use, etc. Since these `#if` statements mostly only test for the existence of a symbol, DICE does a pre-evaluation of the symbols given via the `-D` option, and generates code only for the specified options.

This has the advantage that the generated code is smaller, because it contains only one code snippet with IPC code (for each function) instead of three or four.

This option should only be used if you know that DICE is invoked with the same set of symbols used to compile the generated code.

`test-no-success-message`

The test-suite generally prints a message if a parameter is transmitted correctly and it prints a message if data is not transmitted correctly. With this option only error messages are printed.

`const-as-define`

Constants are usually printed in the generated header files as `const` declarations (`const <type> <var> = <value>;`). Using this option, they are written as `define` statements (`#define <var> <value>`).

`no-l4dir-include-path`

To know the CORBA Types, DICE imports the file `dice/dice-corba-types.h` before parsing the IDL file. The file can (usually) be found in the L4 trees include dir. Therefore the L4 include dir has to be known to include this file.

This directory is automatically added to the include search paths. If you do not wish to set this path, then use the `no-l4dir-include-path` option. The L4 tree's root is determined by the configure switch `--with-l4dir` which defaults to `../...`

`align-to-type`

On some architectures (e.g. ARM) it is necessary to align parameters when marshaling to type size (or word size). To turn this feature on, use the switch `-falign-to-type`. This may waste some space in the message buffer. Since all parameters are sorted by size before marshaling, the padding should be minimal.

`generate-line-directive`

Generates line information in the target code using source file information. So, if a operation was declare on line 10 in the IDL file, the generated functions for this operation will be preceeded be a pre-processor statement containing the IDL file's name and line number of the original declaration.

3.2 Warnings

DICE will print warnings for different conditions if the respective option is given. This section gives an overview of the available warning options. All warning options start with `-W`, the following options have to be added to `-W`.

`ignore-duplicate-fids`

This option will print warnings if there exist duplicate function identifiers within an interface. This will normally cause an error and compilation abort.

`prealloc`

Print warnings if the `malloc` member of the CORBA Environment or the `CORBA_alloc` function are used.

`no-maxsize`

Print warnings if a parameter has no `max_is` attribute assigned to assign a maximum size. DICE uses heuristics to determine its maximum size. Using

this option you may detect parameters which may need an `max_is` attribute to increase or decrease the memory allocate for them.

all

Turns on all of the above warnings.

Appendix A

Pagefault Handling

L4 servers imply special semantics with certain *opcodes*. One is the pagefault IPC. DICE provides some mechanisms to deal with these special semantics.

A.1 L4 version 2

A pagefault IPC consists of two double word values. The first contains the pagefault address and the second the current instruction pointer. Since by convention there are no pagefaults in the micro-kernel's portion of the address space, any pagefault address above 0xC000'0000 is not possible, or handled by the kernel directly (will not involve an IPC). DICE generated code uses the first double word for its opcode.

This knowledge can be used to write a server, which is able to handle pagefault IPC. As described in section 2.5.1, you may specify that the interface identifier and thus the base opcode is 0xC00, which generates opcodes larger than 0xC000'0000.

The server loop now “accepts” only opcodes larger than 0xC000'0000. To handle the pagefaults, one has to specify a call-back function, which is invoked for every “unknown” opcode — which are all opcodes smaller than 0xC000'0000. This is done using the `default_function` attribute for interfaces:

```
[uuid(0xC00), default_function(pf_handler)]
interface pagefault
{
...

```

The call-back function has to have the format:

```
CORBA_int pf_handler(CORBA_Object src,
    <interface>_msg_buffer_t* msg_buffer,
    CORBA_Environment *env)
```

This is basically the format of a normal `<function>_component` function, but instead of parameters it receives the message buffer. The implementation can access the members of the message buffer using the macros described in appendix C. An example implementation of the pagefault handler is provided in appendix B.

The default function has a return value. This is used to determine whether the server loop should respond to the message or not. If you return `DICE_REPLY` the server loop will send a reply message. To skip the reply message, the default function returns `DICE_NO_REPLY`.

A.2 L4 version X.0

The handling of pagefaults in version X.0 is similar to the handling of pagefaults in version 2. A pagefault is here determined by the first double word of a message being zero. The second double word is the pagefault address and the third the instruction pointer. This implies two differences:

1. the server loop determines a pagefault by the opcode 0 and
2. the pagefault handler function has to extract the address and instruction pointer from a different position.

The second problem is solved by using a different implementation of the default function for the two L4 version. The example in appendix B already includes code for version X.0.

The first problem is solved “by accident”. The default function is called for every opcode the server loop cannot match to one of its own opcodes. Since it only knows opcodes starting with `0xC000'0001` and increasing, it does not know opcode 0. Thus it calls the call-back for this opcode as well.

Therefore you may use the same IDL with the shown adaptations in the pagefault handler function to write a pager for L4 version 2 and L4 version X.0.

Appendix B

Pagefault Handler

```
CORBA_int pf_handler(CORBA_Object src,
    handler_msg_buffer_t* msg_buffer,
    CORBA_Environment *env)
{
    l4_umword_t pfa, eip;
    l4_snd_fpage_t page;
    unsigned char ro_or_rw = 0;
    long d;
    // get pfa and eip
#ifdef L4API_14v2
    pfa = DICE_GET_DWORD(msg_buffer, 0);
    eip = DICE_GET_DWORD(msg_buffer, 1);
#else
#ifdef L4API_14x0
    pfa = DICE_GET_DWORD(msg_buffer, 1);
    eip = DICE_GET_DWORD(msg_buffer, 2);
#endif
#endif
    LOG("rcv pf (0x%x, 0x%x)", pfa, eip);
    // check for rw/ro
    if (pfa & 2)
        ro_or_rw = L4_FPAGE_RW;
    else
        ro_or_rw = L4_FPAGE_RO;
    // touch there myself, so I get it
    d = *(long*)pfa;
    LOG("read %d at 0x%x", d, pfa);
    // calculate page (can use src to determine client)
    // (for this example, we simply get the page of the PF-Address
    // and send this page back to the user)
```

```
pfa &= ~0xfff;
page.snd_base = pfa;
page.fpage = l4_fpage(pfa, 12, ro_or_rw, L4_FPAGE_MAP);
// marshal fpage
DICE_MARSHAL_FPAGE(msg_buffer, page, 0);
DICE_SET_SHORTIPC_COUNT(msg_buffer);
DICE_SET_SEND_FPAGE(msg_buffer);
// return that server should reply
return DICE_REPLY;
}
```

Appendix C

Message Buffer Access Macros

<i>Macro</i>	<i>Description</i>
DICE_GET_DWORD Parameters: buf, pos	retrieves the double word at position <code>pos</code> from the message buffer specified by <code>buf</code> <i>Usage:</i> <pre>long dw1 = DICE_GET_DWORD(buffer, 0);</pre>
DICE_GET_STRING Parameters: buf, pos	retrieves the pointer to the indirect part at position <code>pos</code> in the message buffer specified by <code>buf</code> <i>Usage:</i> <pre>char* str = DICE_GET_STRING(buffer, 0);</pre>
DICE_GET_STRSIZE Parameters: buf, pos	retrieves the size of the indirect part at position <code>pos</code> in the message buffer specified by <code>buf</code> <i>Usage:</i> <pre>long str_len = DICE_GET_STRSIZE(buffer, 0);</pre>
DICE_UNMARSHAL_DWORD Parameters: buf, var, pos	retrieves the double word of the position <code>pos</code> from the message buffer <code>buf</code> and stores it in the variable <code>var</code> <i>Usage:</i> <pre>long dw2; DICE_UNMARSHAL_DWORD(buffer, dw2, 2);</pre>

next page ...

... continued from last page

<p>DICE_MARSHAL_DWORD Parameters: buf, var, pos</p>	<p>stores the value of the variable var at the position pos in the message buffer buf. <i>Usage:</i></p> <pre>long variable = 42; DICE_MARSHAL_DWORD(buffer, variable, 3);</pre>
<p>DICE_UNMARSHAL_STRING Parameters: buf, ptr, size, pos</p>	<p>retrieves the pointer to an indirect part and its size from the position pos from the message buffer buf and stores the values in ptr and size respectively. <i>Usage:</i></p> <pre>char *str; long size; DICE_UNMARSHAL_STRING(buffer, str, size, 0);</pre>
<p>DICE_MARSHAL_STRING Parameters: buf, ptr, size, pos</p>	<p>sets the indirect part pointer and size at position pos in the message buffer buf to the values of ptr and size. <i>Usage:</i></p> <pre>char *str = "Hello World!"; long size = strlen(str)+1; DICE_MARSHAL_STRING(buffer, str, size, 0);</pre>
<p>DICE_UNMARSHAL_FPAGE Parameters: buf, snd_fpage, pos</p>	<p>Retrieves the value of a 14_snd_fpage_t structure from the message buffer buf starting at position pos, and stores the values in the snd_fpage variable. <i>Usage:</i></p> <pre>14_snd_fpage_t fpage; DICE_UNMARSHAL_FPAGE(buffer, fpage, 0);</pre>
<p>DICE_MARSHAL_FPAGE Parameters: buf, snd_fpage, pos</p>	<p>Stores the value of the 14_snd_fpage_t structure of variable var in the message buffer buf at position pos. <i>Usage:</i></p> <pre>14_snd_fpage_t fpage; fpage.snd_base = address; fpage.fpage.fpage = other_flexpage; DICE_MARSHAL_FPAGE(buffer, fpage, 0);</pre>

next page ...

... continued from last page

<p>DICE_MARSHAL_ZERO_FPAGE Parameters: buf, pos</p>	<p>Stores zeros at position pos and pos+1 in the message buffer. (L4 uses a "zero flexpage" to separate flexpages from the rest of a message.) <i>Usage:</i> DICE_MARSHAL_ZERO_FPAGE(buffer, 1);</p>
<p>DICE_GET_DWORD_COUNT Parameters: buf</p>	<p>Retrieves the number of send dwords. <i>Usage:</i> num = DICE_GET_DWORD_COUNT(buffer);</p>
<p>DICE_GET_STRING_COUNT Parameters: buf</p>	<p>Retrieves the number of send indirect parts. <i>Usage:</i> str_num = DICE_GET_STRING_COUNT(buffer);</p>
<p>DICE_SET_DWORD_COUNT Parameters: buf, count</p>	<p>Sets the number of send dwords. <i>Usage:</i> DICE_SET_DWORD_COUNT(buffer, num);</p>
<p>DICE_SET_STRING_COUNT Parameters: buf, count</p>	<p>Sets the number of send indirect parts. <i>Usage:</i> DICE_SET_STRING_COUNT(buffer, str_num);</p>
<p>DICE_SET_SHORTIPC_COUNT Parameters: buf</p>	<p>Sets the values of the message buffer to transfer a fast short IPC. It sets the dword count to 2, the indirect part count to 0 and the flexpages to 0. DICE_SET_SHORTIPC_COUNT(buffer);</p>
<p>DICE_SET_SEND_FPAGE Parameters: buf</p>	<p>Sets a flag in the message buffer to tell the reply_any_wait_any function to transfer a flexpage. DICE_SET_SEND_FPAGE(buffer);</p>

Appendix D

CORBA Environment

D.1 L4 version 2 and experimental version X.0

Client side environment

```
typedef struct CORBA_Environment
{
    CORBA_exception_type major:4;
    CORBA_exception_type repos_id:28;
    union
    {
        void *param;
        l4_uint32_t ipc_error;
    } _p;

    l4_timeout_t timeout;
    l4_fpage_t rcv_fpage;
    dice_malloc_func malloc;
    dice_free_func free;
} CORBA_Environment;
```

Server side environment

```
typedef struct CORBA_Server_Environment
{
    CORBA_exception_type major:4;
    CORBA_exception_type repos_id:28;
    union
    {
        void *param;
        l4_uint32_t ipc_error;
    }
}
```

```

    } _p;

    l4_timeout_t timeout;
    l4_fpage_t rcv_fpage;
    dice_malloc_func malloc;
    dice_free_func free;

    // server specific
    void* user_data;
    void* ptrs[DICE_PTRS_MAX];
    unsigned short ptrs_cur;
} CORBA_Server_Environment;

```

D.2 L4 experimental version X.2

Client side environment

```

typedef struct CORBA_Environment
{
    CORBA_exception_type major:4;
    CORBA_exception_type repos_id:28;
    union
    {
        void *param;
        L4_Word32_t ipc_error;
    } _p;

    L4_Time_t timeout;
    L4_Fpage_t rcv_fpage;
    dice_malloc_func malloc;
    dice_free_func free;
} CORBA_Environment;

```

Server side environment

```

typedef struct CORBA_Server_Environment
{
    CORBA_exception_type major:4;
    CORBA_exception_type repos_id:28;
    union
    {
        void *param;
        L4_Word32_t ipc_error;
    } _p;

```

```
L4_Time_t timeout;
L4_Fpage_t rcv_fpage;
dice_malloc_func malloc;
dice_free_func free;

// server specific
void* user_data;
void* ptrs[DICE_PTRS_MAX];
unsigned short ptrs_cur;
} CORBA_Server_Environment;
```

D.3 Linux sockets

Client side environment

```
typedef struct CORBA_Environment
{
    CORBA_exception_type major:4;
    CORBA_exception_type repos_id:28;
    void *param;

    in_port_t srv_port;
    int cur_socket;
    void* user_data;
    dice_malloc_func malloc;
    dice_free_func free;
    void* ptrs[DICE_PTRS_MAX];
    unsigned short ptrs_cur;
} CORBA_Environment;
```

Server side environment

```
#define CORBA_Server_Environment CORBA_Environment
```


Bibliography

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