Bionic Buffalo Tech Note #23

The France IDL Compiler: C Language Mapping

last revised Monday 2003.07.07

Introduction

The `france` OMG IDL compiler generates source code corresponding to the OMG interface definition language (IDL) input. The C mapping is based on the *C Language Mapping Specification* (OMG formal/99-07-35), as extended by Bionic Buffalo. This Tech Note discusses the C source code generated by the compiler, compliance with the *Specification*, ambiguities in the *Specification* and how they were resolved, and the extensions made to the *Specification* by Bionic Buffalo.

The compiler produces code compatible with several different `ORB` implementations. It is intended to be as generic as possible, without sacrificing efficiency.

The information herein corresponds to `france` version A.0.5.38. Previous versions did not agree in some respects. The compiler version has for some time been emitted as a comment in the generated source code.

The Release Flag

The specification requires that there be a release flag for sequences and for structures representing `any` types. The release flag is a boolean value which can be tested or set by certain routines: `CORBA_any_set_release()`, `CORBA_any_get_release()`, `CORBA_sequence_set_release()`, and `CORBA_sequence_get_release()`. However, where the specification describes the structures representing sequences and `any` types, there is no description of a release flag in the mapping. There are two ways to interpret this: the specification gives only partial descriptions of the structures, or the release flags are external to the structures themselves.

The specification requires that the initial value of a structure's release flag ought to be `FALSE`.

In the description of the structures representing sequences, there is an example of a statically-initialized structure, with no initialization of a release flag. To assume a release flag internal to the structure would lead to two difficulties: (1) the complete definition of the structure (with release flag)

would not agree with that in the specification (without a release flag), and (2) the initial value of the release flag might be indeterminate in the example. (Although uninitialized static variables can be assumed to have a value of zero, that assumption is inappropriate for automatic variables. Note that, while the C89 specification did not define initialization of automatic structures, the C99 specification describes such initialization.)

Bionic Buffalo's conclusion is that the release flag must be implemented externally to the structures themselves. Therefore, no `_release` member or similar item is created by the compiler. Note that this practice differs from that of other implementations (such as ORBit, which uses a `CORBA_boolean _release` member), although such differences should make no difference as long as only the specified `CORBA_*_release()` routines are used by applications.

There also is a potential portabilty problem with implementing the release flag as an additional structure member, although it has not yet been seen in practice. On some rarely seen architectures, not all pointers are the same size. (For example, on Univac 1100 hardware, byte addresses consist of word addresses plus byte offsets, although a C compiler might simply use zero offsets for words and force all C pointers to be the same size.) If the `_release` member were put at the beginning of the structure, then the initialization example in the specification wouldn't work correctly. On the other hand, on such machines (with varying-size pointers), putting the `_release` member at the end of the structure representing a sequence would leave its offset unpredictable, since the sequence element type isn't passed to `CORBA_sequence_get_release()`. There are ways around this, but they are worse than the difficulties associated with keeping a release flag external to the sequence structure.

Bionic Buffalo's software, in implementing this decision, makes the assumption that release flags for automatically and statically allocated sequence and `any` structures, initially `FALSE`, cannot be set otherwise. This seems reasonable, in that the only reason for interrogating a release flag is to determine if the secondary storage should be deallocated prior to deallocating the structure itself. In the case of automatic and static structures, there is no reason to free them in the first place.

There is a line of thinking in some discussions that `CORBA_free()` ought to release secondary storage automatically. We see no justification in the specification for this belief. Each secondary buffer (string, sequence buffer, and so on) must be freed separately (by calling `CORBA_free()`, after ascertaining that the release flag, if any, is `TRUE`.) (For convenience, the `yemen` library includes functions to release (recursively) the storage of complex structures, given the structure's address and `TypeCode`.)

Generation of `TypeCode` Constants

The specification requires that an IDL compiler must be able to generate `TypeCode` constants for the input definitions. Therefore, any IDL compiler is necessarily dependent on the implementation's form of `TypeCode`s.

The definition of `TypeCode` structures for Bionic Buffalo's CORBA software is given in the `tibet` header files. Essentially, a `TypeCode` constant is implemented as a `void *` pointer, referencing a

structure whose form varies by the `kind` of definition. The structures are intended to be opaque to applications, understood only by the `CORBA_TypeCode_*()` functions found in the `yemen` library. The various structures (corresponding to different `kind`s of `TypeCode`s or definitions) all have the partial definition

```
typedef  struct tbt_typecode_basic_T
         {
         civ_magic_T      magic ;
         uint_least32_t   flags ;
         uint_least32_t   use_count ;
         unsigned long    kind ;
         ... /* other members, depending on kind */
         }
         tbt_typecode_basic_T ;
```

The `magic` member is set to `TBT_TYPECODE_MAGIC`. One use of the `flags` bits is to specify whether the `TypeCode` structure is statically allocated (by the compiler) or dynamically allocated (by the `ORB`). There are no automatically allocated `TypeCode`s.

For IDL definition `xyz`, the generated `TypeCode` constant is named `TC_xyz`, and the referenced data structure is named `vut_tc_xyz`.

Since `CORBA_free()` is oblivious to secondary storage, and since `TypeCode`s are architecturally a sort of object, storage for `TypeCode`s must be freed using `CORBA_Object_release()`. It is expected that `CORBA_Object_duplicate()` will effectively make a copy of a `TypeCode`. In practice, only the `void *` pointer is duplicated, with a use count maintained for the number of references.

Object References

The specification requires that object references be mapped to `void *` pointers. The compiler makes no assumptions beyond this.

Interface and Operation Information

For each operation defined in IDL, the compiler creates a static data structure of type `tbt_operation_info_T`. This structure contains the information from the IDL regarding the operation and its parameters, and is used by the stubs and skeletons for such purposes as marshalling and unmarshalling of arguments. For operation `oper3` of interface `intf2` (for example), the structure is named `vut_oper_intf2_oper3`.

For each interface defined in IDL, the compiler creates a static data structure of type

`tbt_interface_info_T`. This structure contains the information from the IDL, and includes a table of pointers to the `tbt_operation_info_T` structures for the interface's operations. The structure is used by the stubs and skeletons to match the operation name to a position in the vector of entry point vectors specified by the `POA` mapping to C. (The data structures, as defined by the C mapping in the specification, don't have sufficient information, such as operation names.) For interface `intf2` (for example), the structure is named `vut_intf_intf2`.

Both the `tbt_operation_info_T` and `tbt_interface_info_T` structures are described in the `tibet` headers.

Operation Stubs

Stubs are created for operations defined in IDL. The code in such stubs routes the request to the correct method implementation or remote `ORB`, as appropriate.

The first step taken by a stub is to locate the object, based on its reference. This is done by a function `tbt_target_location()`, supplied by the `ORB` implementation. An object and its methods might be accessible from the calling application in one of several ways.

> *Intrinsic implementation.* The object's method implementation might be directly linked to the application, using the same calling sequence as seen by the application. In other words, the application calls the operation directly, bypassing the `ORB`. In this case, the operation is linked in before the stub, so the stub is neither linked nor called. This situation is seen most commonly in CORBA pseudo-objects, such as `TypeCode`s. There can be only one implementation of each operation in this situation, and that implementation must be local. We need not consider this situation further.
>
> *Linked implementation.* When there is only one local implementation of an operation, and the implementation is written for compatibility with the `POA` mapping, then the implementation can be linked directly to the stub. In this situation, the stub must perform a little housekeeping as well as substitution of the servant for the object reference as the first parameter (since the stub and method signatures differ in the first parameter). Then the stub can call the linked method directly. This mechanism obviates the need to walk the `POA` servant mapping data structures to locate the method.
>
> *Addressable implementation.* When more that one servant may be dynamically registered with the `POA` for a given interface and operation, then the servants are addressable indirectly through pointers found in the servant mapping data structures. There may be more than one local implementation for a given interface. The `tbt_target_location()` function must walk the data structures and return a pointer to the method implementation corresponding to the object reference passed to the stub. After some housekeeping, the stub can call the method indirectly through the method pointer.

> *Remote implementation.* When none of the above situations apply, then the method implementation is not directly addressable in the address space of the application. In this situation, the arguments must be marshalled, a message sent to the implementation, a reply message received, and returned arguments unmarshalled. Although this is called "remote", the situation also applies when the implementation is found in a different process on the same machine. (Note that, on the same machine, some shortcuts may be taken in marshalling arguments.) To effect this, the stub calls a function `tbt_stub_call_remote()`, provided by the `ORB` implementation. The details of the `ORB`'s implementation of this function may vary depending on the `ORB`.

A function's stub will determine which of the above situations applies, then take the steps necessary to invoke the appropriate method implementation. In case of error (such as an invocation on a nonexistent object), the stub will raise an appropriate exception.

An ORB with a dynamic invocation interface or dynamic skeleton interface must be prepared to marshal and unmarshal arguments not only over the wire, but also onto or off from the stack. (Most architectures pass parameters to functions on the stack.) When all interfaces are known at compile time, however, the stack manipulation can be left to the stubs and skeletons. In the case of the stub, before calling the generic `tbt_stub_call_remote()` remote function, the stub builds an array of argument addresses, so the `ORB` developer need not be concerned with the parameter passing mechanism.

On the other end, when arguments must be unmarshalled from the wire and passed to the method, an analogue of the stub is required, which this compiler generates and names (for example) `vut_call_intf2_oper3()` for operation `oper3` on interface `intf2`. These "call" functions take an array of argument addresses and perform an ordinary C function call. The call functions are highly portable. No specialized knowledge about stack structure or other parameter passing mechanisms is required.

Servant Data Structures

For each interface defined in IDL, the compiler produces the `POA` servant data structures required by the specification. For each operation, it produces the source code for a method which returns a `NO_IMPLEMENT` exception. The generated servant data structures reference the generated methods, so any application using the generated structures without modification will receive `NO_IMPLEMENT` exceptions for every operation. The user can modify the data structures as needed to substitute real implementations for the generated methods. To create a real method, the user can start with the generated method as a model. It is recommended that modified servant data structures and methods use modified names, so there will be no conflict or ambiguity when linking with generated code.

The data structures are named as described in the specification. For an operation `oper3` on interface `intf2` (for example), the generated method is named `vut_meth_intf2_oper3()`.

The `_private` member of the interface's entry point vector leads to the above-described `tbt_interface_info_T` structure, and also to the list of call functions (`vut_call_*()`).

For detailed information on the servant data structure as implemented by the `france` compiler, please refer to the `tibet` headers, and consider the generated code in the `tibet` library as examples.

Future Directions

The france IDL compiler generates definitions for insertion into the interface repositories supported by some (but not all) Bionic Buffalo `ORB` implementations. For the other `ORB`s, which do not support dynamic interface repositories, the repository information is not accessible at run time. Bionic Buffalo plans, in a future version of the compiler, to produce data structures in source code form, which will be used to implement a lightweight, read-only interface repository. This lightweight repository will permit applications using any of the `ORB`s to interrogate, but not to modify, any definitions specified in IDL at compile time. The lightweight repository will implement a subset of the same interfaces defined for the full repository.
