Table of contents

1 An introduction to distributed objects and the CORBA bus 1-1
   Objects and object systems 1-1
   Distributed object systems 1-2
   OMG and the CORBA bus 1-3
   CORBA clients and servers 1-5
   Object request broker (ORB) 1-5
   Interface Definition Language (IDL) 1-5
   Language mapping 1-6
   Interface Repository (IR) 1-6
   Static and Dynamic Invocation Interfaces 1-6
   Basic Object Adapter (BOA) 1-7
   Orbix 1-7

2 BAAN ORB Integration: description and architecture 2-1
   Architecture of BAAN ORB Integration 2-1
   BAAN Client architecture 2-2
   BAAN Server architecture 2-3
   Orbix architecture and functionality 2-4
   Activation daemon and the Implementation Repository 2-4
   Locator 2-4
   Interface Repository (IR) 2-5

3 IDL specification and language mapping to Baan C 3-1
   Enabling object interoperability using IDL 3-1
   Structure of a CORBA IDL file 3-2
   IDL lexical conventions 3-4
   Comments 3-4
   Identifiers 3-4
   Keywords 3-5
   Names and scoping 3-6
   Interface declarations 3-7
   Interface header 3-7
   Interface body 3-7
   BAAN C mapping 3-8
   Constant declarations 3-8
   Type declarations 3-9
   Typedefs 3-9
Basic types 3-11
Constructed types 3-14
Template types 3-16
Complex types 3-17
BAAN C specific types 3-17
Table 3-17
Multibyte string 3-17
Object references 3-17
Operation declarations 3-18
Operation attributes 3-19
Operation parameters 3-20
Operation return types 3-20
Attribute declarations 3-21
Parameter passing considerations 3-22
in parameters 3-22
inout parameters 3-22
out parameters 3-23
Return result passing considerations 3-25

4 Creating a BAAN server 4-1
BAAN DLLs as CORBA objects 4-2
Creating a BAAN server: overview 4-4
Step 1: defining the IDL interfaces 4-5
Step 2: creating the DLL
Writing the DLL code: attributes 4-6
Writing the DLL code: operations 4-6
The server DLL 4-7
Registering the IDL interfaces 4-8

5 Creating a BAAN client 5-1
Static and dynamic method invocation 5-1
Writing the BAAN client code: overview 5-3
Step 1: obtaining the object reference 5-4
Step 2: creating and invoking a request 5-4
Accessing the grid attributes 5-4
Setting and retrieving cell values 5-5
Step 3: Releasing the grid object 5-5
The client program 5-6

6 BAAN client API 6-1
Interfaces 6-1
CORBA module 6-1
Functions 6-2
Determining CORBA availability 6-2
Binding to and releasing an object 6-2
Making requests 6-2
CORBA.available() 6-3
Syntax 6-3
Description 6-3
Return value 6-3
CORBA.ORB.string_to_object() 6-3
Syntax 6-3
Description 6-3
Parameters 6-4
Return value 6-4
CORBA.ORB.Release() 6-4
Syntax 6-4
Description 6-4
Return value 6-4
CORBA.Request.invoke() 6-5
Syntax 6-5
Description 6-5
Parameters 6-6
Return value 6-6
Example (two-way) 6-6
Example (one-way) 6-7
CORBA.Request.send() 6-7
Syntax 6-7
Description 6-7
Parameters 6-8
Return value 6-8
Example 6-9
CORBA.Request.get_response() 6-10
Syntax 6-10
Description 6-10
Parameters 6-11
Return value 6-11
CORBA.Request.object 6-11
Syntax 6-11
Description 6-11
Return value 6-12
CORBA.Request.delete 6-12
Syntax 6-12
Description 6-12
Return value 6-12
Mapping of function parameters to ORB elements: overview 6-13
Error codes 6-14

7 Appendix A: Installation and configuration 7-1
   UNIX 7-1
   Step 1 7-1
   Step 2 7-2
   Step 3 7-2
   Step 4 7-2
   Step 5 7-2
   SRV variable: overview 7-2
   BAAN Windows (BW) configuration 7-4
   Windows NT 7-4
   Step 1 7-4
   Step 2 7-4
   BAAN Windows (BW) configuration 7-5

8 Appendix B: Sample IDL definitions and BAAN C client code 8-1
   Supported data types 8-1
   Attributes 8-1
   Operations 8-3
   Operations with parameters of simple type 8-4
   Operations with parameters of type string 8-5
   One-way operations 8-6
   Operations with parameters of type array 8-6
   Object references 8-8
   Sample IDL file and server application 8-9

9 Appendix C: Troubleshooting 9-1
   Server troubleshooting 9-1
   Client troubleshooting 9-2

10 Appendix D: Some server and client limits 10-1
About this document

This manual describes BAAN ORB Integration. This tool enables BAAN servers and clients to communicate with each other and with outside objects using the CORBA object bus.

The manual is intended to provide BAAN developers with all the information they require to create CORBA-enabled BAAN C clients and servers. It assumes that users are familiar with BAAN programming but does not assume any knowledge of CORBA or object-based programming in general.

Summary
Chapter 1 provides an introduction to distributed object systems and to the CORBA object bus.

Chapter 2 describes the architecture of BAAN ORB Integration and provides an overview of Orbix, the ORB implementation used by BAAN ORB Integration.

Chapter 3 describes the OMG IDL specification and the mapping between IDL and BAAN C.

Chapter 4 outlines the procedures for creating a CORBA-enabled BAAN C server.

Chapter 5 outlines the procedures for creating a CORBA-enabled BAAN C client.

Chapter 6 describes the BAAN C API that enables BAAN client applications to access remote objects.

Appendix A provides the information required to install and configure BAAN ORB Integration.

Appendix B presents some sample IDL definitions and BAAN C client code.

Appendix C outlines some of the facilities available for troubleshooting BAAN ORB Integration clients and servers.

Appendix D outlines some of the limitations that apply to BAAN C clients and servers.
### Acronyms

- **ANSI**: American National Standards Institute
- **API**: Application programming interface
- **ASCII**: American Standard Code for Information Interchange
- **BMS**: BAAN Message System
- **BOA**: Basic Object Adapter
- **BW**: BAAN Windows
- **CORBA**: Common Object Request Broker Architecture
- **DII**: Dynamic Invocation Interface
- **DLL**: Dynamic-link library
- **IDL**: Interface Definition Language
- **IEEE**: Institute of Electrical and Electronics Engineers
- **IIOP**: Internet Inter-ORB Protocol
- **IPC**: Interprocess communication
- **IR**: Interface Repository
- **OMG**: Object Management Group
- **ORB**: Object request broker
- **RPC**: Remote procedure call
- **SII**: Static Invocation Interface

### References

1 An introduction to distributed objects and the CORBA bus

This chapter introduces the concept of distributed objects and provides an overview of the OMG CORBA model. This model defines a standard architecture for distributed object systems.

Objects and object systems

An object is a self-contained software entity that provides one or more services that can be used by other objects and applications. One of the key characteristics of an object is that it contains both data and the methods that operate on that data. An object therefore contains all the information required to carry out the services it offers.

An entity using a service is referred to as a client. The program where the object providing the service resides is referred to as the server.

Objects communicate with each other through method calls. A client uses a particular service of an object by invoking a particular method of that object. The server object performs the requested action and returns the result.

Another key characteristic of an object is that it hides its implementation and internal structure from clients; this is referred to as encapsulation. Every object has a public interface that defines the methods that clients use to interact with the object. This interface is all that clients need to know about an object. How the methods are implemented is hidden. Consequently, an object’s internal implementation can change without affecting other objects or applications that use the object’s services.

An object system is a collection of interacting objects that use each other’s services. The objects are independent software entities whose implementations and internal structures are hidden from each other; they communicate with each other solely through method calls.

The benefits provided by object systems include software reusability, portability, and extensibility.

Methods are similar to functions in traditional programming languages. In the OMG Interface Definition Language, they are referred to as operations.

NOTE

Methods are similar to functions in traditional programming languages. In the OMG Interface Definition Language, they are referred to as operations.
Distributed object systems

A distributed system consists of components that are distributed across a network where they communicate with each other remotely.

Distributed object computing results from the convergence of object systems and distributed systems.

In a distributed object system, a collection of distributed objects interoperate as a whole across a heterogeneous network. The objects are packaged as independent software entities that can be accessed transparently using remote method invocations. All a client needs to know about a server object is its interface. The programming language, operating system, and location of a server object is completely transparent to clients (and vice versa).

The following diagram illustrates a distributed object system. The application is distributed across two machines. It consists of a number of interoperating objects contained within client and server applications on the host machines. Note that client and server are relative terms: an object can be both a client and a server, depending on whether it is currently requesting or providing a service.
OMG and the CORBA bus

The Object Management Group (OMG) is an international organization founded to develop, adopt, and promote standards for object-based software in distributed, heterogeneous environments. CORBA (Common Object Request Broker Architecture) was one of the first standards adopted by the OMG. CORBA 2.0, adopted in 1995, extends the original CORBA specification.

CORBA defines a standard, open architecture for distributed object computing. In particular, it specifies the architecture of an object bus (the object request broker or ORB) that links distributed objects such that they can interoperate over a heterogeneous network.

The CORBA object bus is the so-called middleware of distributed object computing. It provides the mechanisms necessary for objects to be able to communicate with one another across the boundaries of heterogeneous languages, platforms, and networks (see the following diagram).
The following are some of the key features and advantages of the CORBA standard:

- **Heterogeneity**: communication between objects on the CORBA bus is completely transparent to those objects. Developers need not be aware of the location, operating system, hardware, or programming languages of remote objects they want to access. Nor do they need to be aware of the mechanisms and protocols used to communicate between objects.

- **Legacy integration**: although CORBA defines standards for object-oriented distributed systems, clients and servers do not have to be written in object-oriented languages. So a BAAN C DLL can participate in a distributed object system, provided that it is furnished with an object-oriented interface.

- **Implementation transparency**: CORBA specifies a standard Interface Definition Language (IDL) for defining the interfaces to CORBA objects. A server object’s implementation is encapsulated behind this formal IDL interface and is thus completely hidden from clients. This enables interoperation between objects written in different programming languages and on different platforms.

The following diagram illustrates the main components of the CORBA object bus. Not all of these are relevant to BAAN developers writing CORBA-enabled clients and servers. For example, Dynamic Interface Invocation (DII) and IDL stubs are alternative mechanisms for remote method invocation and only DII is relevant to BAAN developers. Only those components that are relevant to BAAN developers are discussed further in this chapter.
The following are the main features of CORBA 2.0 that are relevant to BAAN developers:

- CORBA clients and servers
- ORB core
- Interface Definition Language (IDL)
- Language mappings
- Interface Repository (IR)
- Static and Dynamic Invocation Interfaces (SII and DII)
- Basic Object Adapter (BOA)

The following sections provide an overview of each of these features.

**CORBA clients and servers**

A client requests a service from a remote object by invoking an operation on the object. The information associated with a request includes:

- An object reference that identifies the particular object for which the request is intended.
- The name of the method (or operation) to be performed.
- The parameters (if any) of the method.

The server object responds to the request by performing the requested service and returning the results. How the service is implemented is irrelevant to clients.

**Object request broker (ORB)**

The ORB is the core of a CORBA system. It supplies the communication and activation infrastructure for distributed objects on the CORBA bus. This enables clients to transparently invoke methods on a remote server object.

The ORB is responsible for locating and activating an appropriate server object in response to a request from a client, for invoking the appropriate method of the server object, for passing the parameters of the request to the server object, and for returning the results.

**Interface Definition Language (IDL)**

The implementation of a CORBA server object is encapsulated behind a formal interface defined in IDL. Clients view the object purely in terms of this interface.

An IDL interface defines an object’s attributes and the methods it supports (including input and output parameters and their data types). This is all that a BAAN client needs to know in order to make requests on the object. IDL is described in detail in Chapter 3, IDL and language mapping to BAAN C.
Language mapping

IDL is language-independent and purely declarative. An IDL-defined interface can be implemented in any programming language for which a well-defined IDL language mapping is available. This language mapping determines how IDL features map to the facilities of the programming language used to implement an IDL interface.

For example, the language mapping for BAAN C defines how IDL data types map to BAAN C data types. It also defines how IDL interfaces, operations, and attributes map to BAAN C features. BAAN C language mapping is described in detail in Chapter 3, IDL and language mapping to BAAN C.

Interface Repository (IR)

In order to make a request on a CORBA object, a client must know the IDL interface(s) of that object. Object definitions are therefore stored in a repository (the IR) which is accessible to other objects on the CORBA bus. From the repository, clients can obtain the full definitions of object interfaces and their associated attributes, operations, and so on.

Static and Dynamic Invocation Interfaces

A client uses the services of a CORBA object by invoking the object’s methods. CORBA defines two invocation interfaces: Static Invocation Interface (SII) and Dynamic Invocation Interface (DII).

The SII consists of client IDL stubs that are generated by an IDL compiler. To make a request, the client calls a stub routine that issues the request on the client’s behalf. The routine is specific to a particular operation on a particular object and is compiled into the client application. This limits the client to using only those services known to and selected by the client programmer when building the application.

The DII is more complex but also more flexible. It provides a generic way to call a server that allows an application to issue requests for any interface. The interface does not have to be known at the time the application is compiled.

Using the DII, a client program can invoke a remote object by specifying, at runtime, the target object reference, the operation/attribute name, and the parameters to be passed. DII is the invocation method used by BAAN C clients. It is described in more detail in Chapter 5, Creating a BAAN client.
Basic Object Adapter (BOA)

An object adapter is the primary interface that an object implementation uses to access the services of an ORB; it is not used by CORBA clients. Object adapters work closely with the programming languages in which objects are implemented. Consequently, they tend to be language-specific. However, the CORBA-defined BOA is a general-purpose object adapter that can be used for most CORBA objects with conventional implementations. In the BAAN ORB Integration product, the BOA is implemented by the ORB Server component (see Chapter 2, BAAN ORB Integration: description and architecture).

Orbix

CORBA 2.0 specifies a standard architecture for an ORB. To build a CORBA system, an actual ORB implementation is required. The ORB implementation chosen as the foundation for BAAN ORB Integration is Orbix from IONA Technologies Ltd (for further details, see Chapter 2, BAAN ORB Integration: description and architecture).
An introduction to distributed objects and the CORBA bus
OMG’s CORBA specification defines a standard for distributed object computing. The core of a CORBA-compliant system is the object request broker (ORB). This provides the basic mechanisms that enable interaction between distributed objects.

The ORB enables objects to transparently send and receive requests and responses in a distributed environment. It is responsible for:

- Locating and activating objects in response to a request from a client.
- Invoking the appropriate method of the server object.
- Passing the parameters of a request to the server object.
- Returning the results.

The ORB is the foundation for building applications from distributed objects and for interoperability between applications in heterogeneous environments.

This chapter describes the architecture of BAAN ORB Integration, the tool that enables BAAN servers and clients to communicate with each other and with outside objects using the CORBA object bus. It also provides an overview of Orbix, the ORB implementation used by BAAN ORB Integration to handle communications between remote clients and servers. Orbix is a product of IONA Technologies Ltd.

**Architecture of BAAN ORB Integration**

BAAN ORB Integration provides facilities for the exchange of data between distributed BAAN programs and between BAAN programs and external objects. It supports direct calls to remote objects from BAAN C programs and direct calls from remote objects to BAAN C DLLs. It can be viewed as a general communication gateway between BAAN's ERP application and remote objects.

BAAN ORB Integration consists of two main products:

- The BAAN Client
- The BAAN Server

These two products can be used totally independently of each other. However, a BAAN application can be both a server and a client at the same time; that is, it can both provide services to, and use the services of, other CORBA objects.
The following diagram provides an overview of the architecture of the BAAN ORB Integration product.

**BAAN Client architecture**

A CORBA client is an object that uses the services of a remote server object by issuing requests to invoke the server’s methods (or operations).

The BAAN Client product provides the mechanisms that enable BAAN C applications to act as CORBA clients. The BAAN C application does not need to know the server object’s location, operating system, or programming language. The connection to a server is completely transparent to the client.
The BAAN Client product is a BAAN bshell extended with two APIs: Request and ORB. These APIs are implementations of the CORBA-defined Request and ORB interfaces respectively. They enable BAAN C client applications to connect to and make requests on remote CORBA objects. The APIs work in conjunction with the BAAN Message System (BMS) to enable BAAN C clients to issue deferred synchronous calls to CORBA objects.

Chapter 6, BAAN client API, describes in detail the Request and ORB APIs and the use of the BMS to make deferred synchronous calls.

**BAAN Server architecture**

A CORBA server is a program that holds objects which provide services to other CORBA objects. It responds to requests from a client object for a service by executing the appropriate methods (or operations) and returning the results. The services offered by an object are defined in the CORBA Interface Definition Language (IDL). They can be implemented using any programming language for which an IDL language mapping is available. The implementation is completely transparent to clients; clients need only be aware of the IDL definition.

The BAAN Server product provides the mechanisms that enable BAAN C DLLs to act as CORBA servers and to make their services available to CORBA clients. The client does not need to know the server's location, operating system, or programming language. The connection to the server is completely transparent. Any CORBA-enabled application can connect to a BAAN server and use its services.

The BAAN Server product consists of the ORB Server which translates requests from CORBA clients into BAAN C DLL calls. It also implements the Basic Object Adapter defined by CORBA (see Chapter 1, An introduction to distributed objects and the CORBA bus). There are two parts to the ORB Server:

- A process, external to the bshell, that communicates with the ORB and translates ORB requests into IPC (interprocess communication) requests.
- A BAAN C object (that is, a bshell process) that receives the IPC requests from the external process, translates them into BAAN C DLL calls, and returns the results as an IPC reply.
A CORBA-enabled BAAN C server therefore consists of:

- The ORB Server which translates requests from CORBA clients into BAAN C DLL calls. This server also translates the replies into an appropriate format for delivery by the ORB.
- A BAAN bshell which is the application server for BAAN CORBA objects.
- The BAAN CORBA server object which is a BAAN C DLL.

Chapter 4, Creating a BAAN server, describes the procedures for creating a BAAN C server.

Orbix architecture and functionality

The OMG CORBA standard specifies the architecture of an ORB but stipulates few implementation requirements. An ORB may be implemented, for example, as a set of runtime libraries, as a set of daemon processes, or as part of an operating system.

Orbix is implemented as a pair of libraries (one for client applications and one for servers) and an activation daemon.

Activation daemon and the Implementation Repository

The role of the activation daemon (orbixd) is to launch an appropriate server in response to a client request and to establish the connection between the client and the server. It is not involved in message transmission.

The daemon uses a simple database, the Implementation Repository, to identify which executable file launches a particular server. All servers must therefore be registered with the Implementation Repository. The repository then provides a mapping from the registered server name to the associated executable file. Objects within a BAAN server (that is, a bshell) are registered dynamically at the moment a request is received.

Locator

When a client wishes to use a remote service, it instructs the ORB to establish a connection to a suitable server. If the client does not specify the name of the host on which the required server resides, the Locator is used to search for the server.
The Locator maintains a list of all registered server names and the locations at which those servers are available. When searching for a server, the Locator requests from the local Orbix daemon a list of the hosts on which the required server resides. It then searches through this list until it finds a host at which the server is registered.

Note that the Locator is an Orbix concept and not a CORBA one.

**Interface Repository (IR)**

In order for an ORB to correctly process requests, it must have access to the definitions of the objects it is handling. The IR is one means by which object definitions are made available both to the ORB and to clients and servers.

The IR maintains full information about the IDL definitions of object interfaces. BAAN programmers must register IDL definitions of objects they have created with the IR, if they wish those definitions to be publicly available.

Other programs can then use the IR to find the full definitions of specific interfaces, attributes, operations, and so on. These programs can also browse through the set of modules and interfaces known to Orbix and determine their names and interface definitions.
Objects on the CORBA bus can interoperate across programming languages, operating systems, and networks. To facilitate this interoperability, CORBA objects have clearly defined interfaces that are language independent and implementation independent. These interfaces are defined using OMG's Interface Definition Language (IDL).

The IDL definition of an object describes the functionality (or services) that the object offers to clients: that is, the methods that a client can invoke on the object and the parameters and return values of those methods. The IDL definition is purely descriptive: it provides no implementation details. How an object’s services are implemented is completely transparent to clients. How a client is implemented is equally transparent to servers.

When the interface to an object has been defined in IDL, the object can be implemented in any appropriate programming language, and clients can use any language to make remote requests on the object. In particular, the programming language used to implement an object can differ from that used to implement a client that uses the object.

This chapter describes the OMG IDL specification and the mapping between IDL and BAAN C.

**Enabling object interoperability using IDL**

To enable CORBA objects written in different programming languages to interoperate:

- The IDL definition of an object must fully specify that object’s interface: its attributes, methods, parameters, return values, data types, and so on.
- The server code must implement the IDL definition exactly.
- The IDL definition of an object must be available to clients via an Interface Repository (see Chapter 1, An introduction to distributed objects and the CORBA bus, for further details).
- Mappings from IDL must have been defined for both the server and the client programming languages.
Clients can interact with a CORBA object using either of the following invocation interfaces:

- The Static Invocation Interface (SII), which supports direct mapping between the client language and IDL.
- The Dynamic Invocation Interface (DII), for which no direct mapping is required.

BAAN C clients use the DII (see Chapter 5, Creating a BAAN client, for further details).

**Structure of a CORBA IDL file**

The IDL specification of a CORBA object is contained in a file with the extension `.idl`. The specification of any object interface consists of one or more definitions of the following elements:

- Modules
- Interfaces
- Attributes
- Operations
- Data types
- Constants
- Exceptions

Modules and exceptions are not supported by BAAN C and are not discussed further in this chapter. All other elements are discussed in separate sections of this chapter.
The following diagram illustrates the structure of an IDL file and the main elements applicable to BAAN C.

An interface is the basic unit of an IDL definition. It defines the operations that a client can invoke on an object. An IDL interface maps to a BAAN C DLL of the same name. The DLL implements the services defined in the interface.

An operation defines a particular service that an object offers. The definition includes specification of the parameters and return values of the operation and their data types. An operation maps to a BAAN C external function.

An interface may also define attributes for an object. These are values that map to _get_ and _set_ functions in BAAN C.
IDL lexical conventions

This section describes the lexical conventions of OMG IDL.

Comments

IDL supports two comment formats:

// comment

Comments in this format terminate at the end of the line on which they occur. The characters //, /*, and */ have no special meaning within a comment of this type; they are treated as ordinary characters.

/* comment */

Comments in this format can span more than one line. The characters // and /* have no special meaning within a comment of this type; they are treated as ordinary characters.

Comments may contain alphabetic, numeric, graphic, space, horizontal tab, vertical tab, form feed, and new line characters.

There is no mapping of IDL comments to BAAN C.

Identifiers

An IDL identifier is a name string consisting of a sequence of alphanumerical and underscore (_) characters. The first character must be alphabetic.

IDL identifiers are case-insensitive in that you cannot define two identifiers that differ only in case. For example, you cannot define two different interfaces named Grid and grid.

IDL identifiers must be spelled consistently, with respect to case, throughout a specification. For example, the same interface cannot be referred to in one instance as Grid and in another as grid. This holds true across file boundaries also; identifiers in different files match only when their cases match. This restriction is imposed in order to support natural mappings to case-sensitive languages.
When comparing two identifiers to determine if they collide, the following rules apply:

- Uppercase and lowercase letters are treated as the same letter. That is, identifiers that differ only in case collide.
- Digraphs and pairs of letters (for example, æ and ae) are not considered equivalent.
- Accented and nonaccented letters (for example, a and à) are not considered equivalent.
- All characters are significant.

IDL identifiers differ from BAAN C identifiers only in that they cannot contain a period [.] and are case sensitive (BAAN C identifiers are not). IDL identifiers are therefore a subset of BAAN C identifiers and map directly to them.

**Keywords**

The following identifiers are reserved as IDL keywords, and must not be used otherwise:

<table>
<thead>
<tr>
<th>any</th>
<th>default</th>
<th>inout</th>
<th>out</th>
<th>switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>double</td>
<td>interface</td>
<td>raises</td>
<td>TRUE</td>
</tr>
<tr>
<td>boolean</td>
<td>enum</td>
<td>long</td>
<td>readonly</td>
<td>typedef</td>
</tr>
<tr>
<td>case</td>
<td>exception</td>
<td>module</td>
<td>sequence</td>
<td>unsigned</td>
</tr>
<tr>
<td>char</td>
<td>FALSE</td>
<td>object</td>
<td>short</td>
<td>union</td>
</tr>
<tr>
<td>const</td>
<td>float</td>
<td>octet</td>
<td>string</td>
<td>void</td>
</tr>
<tr>
<td>context</td>
<td>in</td>
<td>oneway</td>
<td>struct</td>
<td></td>
</tr>
</tbody>
</table>

Keywords obey the rules for identifiers and must be written exactly as shown in the above list. For example, 'boolean' is correct; 'Boolean' is not.

IDL specifications use the characters ; { } : , = + − ( ) < > [ ] ’ " \ | ^ & * / % ~ as punctuation. The tokens # ## ! || && are reserved for use by the IDL preprocessor.
Names and scoping

An IDL file forms a naming scope in which an identifier is defined and can be referred to. Other IDL elements also form distinct naming scopes; those relevant to BAAN C are file, interface, and operation.

Every IDL identifier must be unique within the current scope. An identifier, however, can be reused within other distinct scopes outside the current scope.

For example, an interface forms a distinct scope within an IDL file. A name declared within an interface is visible only in that interface and any other scopes nested within the interface (for example, operations). Consequently, if the same names are used within different interfaces, those names do not collide.

In the following IDL example, the operation `xyz` in the `alpha` interface has the same name as the operation `xyz` in the `beta` interface. However, as the two operations reside in different, distinct scopes, their names do not collide.

```idl
// file example.idl
interface alpha {
    enum type { one, two, three }
    operation xyz ( )
}
interface beta {
    operation xyz ( )
}
```

A scoped name has the form `<scoped_name>::<identifier>`. Within a scope, a name may be used in its unqualified form.

The fully scoped IDL names for the operations in the example above are `::alpha::xyz` and `::beta::xyz`. The double colon `::` is the scope resolution operator. The first double colon indicates the file scope. The field `one` from the enumerator `type` is referenced as `::alpha::type::one` from anywhere outside the interface `alpha`, and as `type::one` within the interface.

The IDL scope resolution operator maps to a period `[.]` in BAAN C. An underscore followed by a full stop `[_.]` indicates the file scope. BAAN C does not have a scoping mechanism other than the file and the function. As a result, fully scoped names must be used in BAAN C in order to avoid name clashes; also the root scope operator is always omitted from BAAN C names. Therefore, the fully scoped BAAN C names for the operations in the above example are `alpha.xyz` and `beta.xyz`. These fully scoped names must always be used to refer to the operations. Similarly, the field `one` is always referenced as `alpha.type.one` in BAAN C, whether from inside or outside the operation `alpha`. 
Note that in BAAN C, a period [.] has no special meaning; it is a normal character and is seen as part of the name. However, the period is not a valid character for names in IDL. So it is interpreted as the scope resolution operator by the BAAN ORB implementation.

For a full discussion of IDL scoped names, consult the Common Object Request Broker Architecture and Specification.

### Interface declarations

All interfaces must be defined at global scope: that is, nested interfaces are not supported. An interface declaration has two parts: the interface header and the interface body.

#### Interface header

The interface header consists of the interface identifier preceded by the keyword `interface`. For example:

```idl
// IDL
interface an_interface_name {
        . . .
};
```

#### Interface body

The interface body consists of a series of declarations of the following kinds:

- Constant declarations, which specify the constants that the interface exports.
- Type declarations, which specify the type definitions that the interface exports.
- Exception declarations, which specify the exception structures that the interface exports (exceptions are not supported by BAAN C).
- Attribute declarations, which specify the attributes exported by the interface.
- Operation declarations, which specify the operations that the interface exports and the format of each.

These declaration types are described in separate sections of this chapter.
BAAN C mapping

There is no mapping for BAAN C clients. These use the DII instead (see Chapter 5, Creating a BAAN client).

On the server side, an IDL interface maps to a BAAN C DLL of the same name. The DLL implements the services defined in the interface. However, because a BAAN DLL resides in a package and module, the IDL interface name must be a concatenation of the package, module, and library names.

For example, take a DLL named `bank`, that is created in the module `tst` in the package `tc`. The DLL file name is `tctstbank`. The IDL name for the interface is also `tctstbank` and is declared as follows:

```idl
// IDL
interface tctstbank {
    . . .
};
```

Multiple interfaces may be defined within the same IDL file. However, it is recommended that each IDL file contains only a single interface definition, and that the IDL file and interface take the same name as the DLL.

Note that BAAN C does not support derived interfaces.

Constant declarations

In IDL, a constant is an invariable value with a type and a name. The type can be integer (short, long, and so on), char, boolean, float, double, or string. By convention, constant names are in uppercase. Constants can be declared within an interface or at file level scope, as appropriate. For example:

```idl
// IDL
const string BANKNAME = "Borabank";
// IDL
interface bank {
    const string BANKNAME = "Borabank";
    . . .
}
```
IDL constants map to a constant definition in BAAN C. BAAN C constants differ from IDL constants in that they do not have an associated type.

The IDL constant defined above is declared in BAAN C as follows:

```c
#define BANKNAME "Borabank"
#define bank.BANKNAME "Borabank"
```

### Type declarations

Data types are used to describe the acceptable values of CORBA parameters, attributes, exceptions, and return values. CORBA supports four categories of types: basic, constructed, template, and complex. Each of these categories is described in a separate section below.

### Typedefs

IDL provides a mechanism for associating meaningful identifiers with data types. A C language-like declaration, with the `typedef` keyword, is used to define the identifier for a type. Typedefs are not supported in BAAN C. However, interfaces that use typedefs can be used by or implemented in BAAN C if the underlying data types are supported.

In the following IDL example, a `typedef` declaration defines `Name` as a synonym for `string<20>`. Since strings are supported by BAAN C, the typedef can be mapped to BAAN C.

```idl
// IDL
typedef string<20> Name;
interface UserAdmin {
    attribute Name User;
};
```

```c
string username(20)
function extern void UserAdmin._set_User(const string s())
function extern string UserAdmin._get_User()
```
In the next example, a typedef declaration defines `Engineer` as a synonym for the `Person` structure. BAAN C does not support structures and so the interfaces that use the typedef cannot be used by or implemented in BAAN C.

```c
// IDL - no mapping to BAAN C
struct Person {
    long pin;
    string<20> name;
};
typedef Person Engineer;
interface personnel {
    Engineer FindEngineer ( in long pin );
    long FindPin ( in Engineer e );
};
```

It is recommended that you do not use typedefs for interfaces that are to be implemented in BAAN C, even if the underlying data types are supported.
**Basic types**

The following table lists and describes all CORBA basic types and indicates how they map to BAAN C types. A hyphen [–] in the third column indicates that the type is not supported by BAAN C.

<table>
<thead>
<tr>
<th>CORBA type</th>
<th>Description</th>
<th>BAAN C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>Integer values: -32,768 through 32,767 (-2¹⁵ through 2¹⁵ - 1)</td>
<td>long</td>
</tr>
<tr>
<td>long</td>
<td>Integer values: -2,147,483,648 through 2,147,483,647 (-2³¹ through 2³¹ - 1)</td>
<td>long</td>
</tr>
<tr>
<td>unsigned short</td>
<td>Integer values: 0 through 65,535 (0 through 2¹⁶ - 1)</td>
<td>long</td>
</tr>
<tr>
<td>unsigned long</td>
<td>Integer values: 0 through 4,294,967,296 (0 through 2³² - 1)</td>
<td>long</td>
</tr>
<tr>
<td>float</td>
<td>Single precision IEEE 754 floating point number</td>
<td>double</td>
</tr>
<tr>
<td>double</td>
<td>Double precision IEEE 754 floating point number</td>
<td>double</td>
</tr>
<tr>
<td>char</td>
<td>A single ISO Latin-1 (8859.1) character</td>
<td>string(1)</td>
</tr>
<tr>
<td>boolean</td>
<td>Truth value (1=TRUE or 0=FALSE)</td>
<td>long</td>
</tr>
<tr>
<td>octet</td>
<td>8-bit quantity guaranteed not to undergo any conversion during transmission</td>
<td>long</td>
</tr>
<tr>
<td>any</td>
<td>A type that can represent any CORBA type</td>
<td>–</td>
</tr>
</tbody>
</table>

In the following IDL example, a data item (overdraftLimit) to be used by a bank interface is declared as type float. Also, a flag that indicates whether or not an account is locked is declared as type boolean.

```idl
// IDL
float overdraftLimit;
boolean locked;
```

In BAAN C, the following code implements these declarations:

```baan
| BAAN C
double overdraftLimit
long locked
```

Note that because `locked` is declared as `boolean` in IDL, the `long` data item declared in BAAN C can have only the values 1 (TRUE) or 0 (FALSE).
Mapping considerations

For some mappings from IDL data types to native BAAN C data types there is a risk of data loss, and programmers should be particularly careful when using these types.

For example, care should be taken that no overflow takes place when a short value is converted to a long and is later converted back to a short. Also, the mapping for unsigned long is particularly unsafe as it may result in overflow from IDL to BAAN C.

When defining IDL that is to be implemented in a BAAN C server, it is preferable to use only the types that safely map to BAAN C. These types are: long, double, and string.

The client invocation mechanisms CORBA.Request.invoke() and CORBA.Request.send() return an error code to the client if an attempt is made to convert an illegal value.

Integer type

OMG IDL supports long and short, signed and unsigned integer data types. The value ranges represented by each type are listed in the table in the section Basic types.

All IDL integer types map to a long in BAAN C. For some mappings there is a risk of data loss. For example, BAAN C long to CORBA short or CORBA unsigned long to BAAN C long. Programmers should take care to avoid data loss when using these types.

Floating-point type

OMG IDL defines two floating-point types: float and double. These represent IEEE single-precision floating-point numbers and IEEE double-precision floating-point numbers respectively. For further information, consult the IEEE Standard for Binary Floating-Point Arithmetic.

Both floating-point types map to a double in BAAN C. There are only two constraints: in BAAN C, double literals must be specified with a decimal point, and the exponent part of the double must be within the range -307 .. 307.
The following example illustrates how an IDL-defined constant `float` value is defined in BAAN C.

```c
// IDL
const float f1 = 1e10;
| BAAN C
#define f1 1.0e10
```

**Char type**

OMG IDL defines a `char` data type consisting of 8-bit quantities. The ISO Latin-1 (8859.1) character set standard defines the meaning and representation of all possible graphic characters (that is, the space, alphabetic, numeric, and graphic characters). The meaning and representation of the NUL and formatting characters (BEL, BS, HT, NL/LF, VT, FF, CR) is the numerical value of the character as defined in the ASCII (ISO 646) standard. The meaning of all other characters is implementation dependent.

During transmission, characters may be converted to other appropriate forms. Such conversions may change the representation of a character but must maintain the character's meaning. For example, a character can be converted to and from the appropriate representation in international character sets, according to the machine locale.

A `char` is mapped to a `string` of length one in BAAN C. The content of the string is a single character coded as defined in ISO Latin-1. An empty string represents the NUL character. For example:

```c
// IDL
char s;
| BAAN C
string s(1)
```

**Boolean type**

The `boolean` data type is used to denote a data item that takes only the values TRUE and FALSE. A `boolean` maps to a BAAN C `long` that must hold only the values 0 (FALSE) or 1 (TRUE). According to the CORBA specification for IIOP (Internet Inter-ORB Protocol), an application should hard-code the values 0 and 1 instead of using symbolic constants.
Octet type

The octet type is an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system. An octet maps to a BAAN C long that takes the values 0..255, according to the decimal translation of the binary representation of the octet.

Constructed types

OMG IDL supports three constructed types: struct, union, and enum. BAAN C supports only enumerations.

Enumeration type

Enumerated types consist of ordered lists of identifiers. A maximum of $2^{32}$ identifiers may be specified in an enumeration. The order in which the identifiers are named in the IDL specification defines the relative order of the identifiers.

An IDL enum maps to a long in BAAN C; the possible values for it are defined using a #define statement. The first identifier must be defined as zero; succeeding identifiers take ascending numbers in the order of the declaration (from left to right).

The following example illustrates how an IDL enum type maps to BAAN C.

```
// IDL
enum anEnum { one, two, end};
anEnum myEnum;

| BAAN C |
#define anEnum.one 0
#define anEnum.two 1
#define anEnum.end 2
long myEnum
```

IDL enumerations differ from BAAN C enumerate domains in two ways:

- BAAN C enumerates have values that start with 1, while IDL enums start with 0.
- BAAN C enumerate values are not guaranteed to be successive, while IDL enum values are successive.

However, an existing BAAN enumeration can be reflected in IDL by using dummy entries in the IDL specification.
For example, the following illustration shows the definition of an enumerate domain named \texttt{tcosys} in BAAN.

This can be defined in IDL as follows:

```idl
// IDL
enum tcosys { dummy1, sic, mps, mrp, fas, dummy2, dummy3, mnl };
```

When the IDL specification is mapped back to BAAN C, the result is:

```c
| BAAN C
#define tcosys.dummy1 0
#define tcosys.sic 1 | or sic
#define tcosys.mps 2 | or mps
#define tcosys.mrp 3 | or mrp
#define tcosys.fas 4 | or fas
#define tcosys.dummy2 5
#define tcosys.dummy3 6
#define tcosys.mnl 7 | or mnl
domain tcosys myEnum | equivalent to long
```
Existing implementations can use the original BAAN values, without the prefix, instead of the CORBA-defined values; for example, domain tcosys and sic. New implementations derived from the IDL specification would use long and tcosys.sic. However, it is important that the two implementations are compatible.

**Template types**

OMG IDL supports two template types: sequence and string. BAAN C supports only strings.

**String type**

A string is a sequence of any characters; only the NUL character (\0) cannot be included in the sequence.

IDL distinguishes between bounded and unbounded strings. A string is bounded if the string declaration includes a parameter specifying the maximum size of the string; otherwise it is unbounded. For example:

// IDL - bounded string
string<20> Name;

// IDL - unbounded string
string Name;

IDL bounded strings map directly to BAAN C strings not declared as based. IDL unbounded strings map to based strings in BAAN C.

For example, the two IDL strings defined above are declared in BAAN C as follows:

| BAAN C - bounded string
string Name(20)

| BAAN C
string Name based

When an unbounded string is used, client applications must explicitly allocate memory for it using alloc.mem(). For clients, this implies a maximum size of 5 MB for unbounded strings. For servers, the maximum size for strings of each type is 16K.

Strings are often used as parameters or return values of operations. In such cases, the mapping is different (for details, see the sections Parameter passing considerations and Return result passing considerations later in this chapter).
Complex types

Array type

OMG IDL defines multidimensional fixed-size arrays to hold lists of elements of the same type. The size of each dimension is specified in the definition. IDL arrays map to BAAN C arrays. For example:

```c
// IDL 3-dimensional array
short grid[10] [20] [15]
```

<table>
<thead>
<tr>
<th>BAAN C 3-dimensional array</th>
</tr>
</thead>
<tbody>
<tr>
<td>long grid(10,20,15)</td>
</tr>
</tbody>
</table>

On the client side, arrays of `long` and `double` are supported. They can have up to four dimensions. Arrays can be used as parameters or return values of operations, but not as attributes. When an array is passed as a parameter in an operation invocation, all elements of the array are transmitted.

On the server side, there is no mapping for IDL arrays.

BAAN C specific types

Table

Table variables are not supported by CORBA.

Multibyte string

In BAAN C, TSS strings provide multilanguage support. A TSS string must be converted to ISO Latin-1, according to the locale of the local machine, before it is passed via CORBA. It is passed in a normal string.

Object references

An object reference is a name or identifier that uniquely identifies an object within a CORBA system.

Object references map to a `long` in BAAN C. The `long` holds a unique id. An object reference is created by calling `CORBA.ORB.string_to_object()`.

BAAN servers do not support object references. BAAN clients can use object references either in the operation return value or in operation parameters.
The following IDL specification defines an interface named `account`. When a BAAN C client binds to an object that implements this interface, an object reference is returned for the newly-created account object.

```idl
// IDL
interface account {
    readonly attribute float balance;
    ...
};
```

```baan-c
long objref
objref = CORBA.ORB.string_to_object( ..., "account", ... )
```

## Operation declarations

An operation (also referred to as a method or function) defines a particular service that an object offers to clients. An IDL operation declaration is similar to a BAAN C function declaration. It consists of the following elements:

- An optional operation attribute, which specifies which invocation semantics the communication system should provide when the operation is invoked.
- The type of the operation’s return result.
- An identifier, which names the operation in the scope of the interface in which it is defined. This maps directly to a BAAN C identifier (see the section Identifiers earlier in this chapter).
- A parameter list, which defines zero or more parameters for the operation.

For example, the following IDL segment declares two operations (`makeDeposit` and `findBalance`) in the interface `ttstpaccount`.

```idl
// IDL
interface ttstpaccount {
    oneway void makeDeposit ( in float sum );
    float findBalance ( out string nameAddress );
};
```
These map to two external functions in a BAAN C server. Each function name is a concatenation of the interface name and the function name. For example

```c
function extern void ttstpaccount.makeDeposit (const double sum)
function extern double ttstpaccount.findBalance (ref string nameAddress)
```

There is no mapping for BAAN C clients. These use the DII to call operations.

**Operation attributes**

This optional element of an operation declaration specifies the invocation semantics that the client can expect from a request for the operation.

Currently there is one possible operation attribute: `oneway`. This specifies a request-only operation; the operation does not return any results and the client is not blocked while the request is being processed.

For a one-way operation, the invocation semantics are `best-effort`. Delivery of the call is not guaranteed, and the operation will be invoked at most once.

The following is an example of one-way operation declaration:

```c
// IDL
oneway void signalError ( void );
```

The operation must not include any output parameters and must specify a `void` return type.

By default, all operations are two-way operations (this is not declared explicitly). In this case, the invocation semantics are `at-most-once` if an exception is raised and `exactly-once` if the call returns successfully.

Both one-way and two-way operations map to BAAN C external functions. There is no difference in the way they are coded; however, whether an operation is one-way or two-way does influence how it is called.
Operation parameters

An operation parameter is characterized by its mode and its type. The type constrains the possible values of the parameter (data types are discussed in detail earlier in this chapter). The mode indicates the direction in which the parameter value is passed. The possible directional modes are:

- **in** – the parameter is passed from client to server. An in parameter maps to a const parameter in BAAN C.
- **out** – the parameter is passed from server to client. An out parameter maps to a ref parameter in BAAN C.
- **inout** – the parameter is passed in both directions. An inout parameter maps to a ref parameter in BAAN C.

A server receiving an in parameter may not modify it. In BAAN C, this is enforced by declaring it as a const. Also, when an unbounded string is passed as an inout parameter, the returned value must not be longer than the input value.

The following example indicates how IDL operations map to BAAN C: // IDL

```idl
interface ttstpaccount {
    void makeDeposit ( in float deposit, inout float balance, out short error );
};
```

<table>
<thead>
<tr>
<th>BAAN C</th>
</tr>
</thead>
<tbody>
<tr>
<td>function extern void ttstpaccount.makeDeposit ( const double deposit, ref double balance, ref long error )</td>
</tr>
</tbody>
</table>

For information on memory management for parameters, see the section Parameter passing considerations, later in this chapter.

Operation return types

The type of an operation return value in IDL maps to BAAN C in the same way as the type of a parameter. Theoretically, there is no difference between parameters and return values; a return value is simply an out parameter written in a different notation. For example, both the following statements return \( x \), the first returns it in an out parameter, the second returns it as a return value:

<table>
<thead>
<tr>
<th>BAAN C (out parameter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x, . . .) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BAAN C (return value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = f(. . .) )</td>
</tr>
</tbody>
</table>
A return value has no explicit mode; it is always `out`. A special return type `void` indicates the absence of a return value. All basic types can be returned by an operation.

**Attribute declarations**

Any object can have one or more attributes. These are properties of the object that can be accessed by clients. A program can retrieve the attributes of a local object by accessing the memory locations where they are stored. However, this is not possible for remote objects, as the memory locations where the attributes are stored reside elsewhere on the network.

To enable clients to access object attributes transparently over the network, CORBA defines two accessor functions for each attribute:

- `_get_`, which retrieves the value of the attribute.
- `_set_`, which sets the value of the attribute.

An IDL attribute definition is therefore logically equivalent to declaring a pair of accessor functions. The optional `readonly` keyword indicates that there is only a single accessor function: that is, to retrieve the value of the attribute. Clients call the `_get_` and `_set_` accessors as if they were operations.

In the following example, the attribute `overdraftLimit` is defined as a property of the `tctstbank` interface. In the BAAN C implementation, two external functions are declared to get and set the values of this attribute.

```idl
interface tctstbank {
    attribute float overdraftLimit;
    ...  
}

<table>
<thead>
<tr>
<th>BAAN C</th>
</tr>
</thead>
<tbody>
<tr>
<td>double overdraftLimit</td>
</tr>
<tr>
<td>function extern double tctstbank._get_overdraftLimit ( void )</td>
</tr>
<tr>
<td>{ return ( overdraftLimit )</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>function extern void tctstbank._set_overdraftLimit ( const double limit )</td>
</tr>
<tr>
<td>{ overdraftLimit = limit</td>
</tr>
</tbody>
</table>
| }

Note that CORBA defines only the two accessor functions. Declaring a local variable with the name `overdraftLimit` is therefore not a requirement.
However, it is usual to declare a DLL global variable to act as the placeholder for the actual attribute value. To ensure that the variable is not visible outside the DLL, the global variable is not declared as external. Note also that it is a good programming practice to use the accessor functions even within the DLL itself.

**Parameter passing considerations**

When writing BAAN C code that uses or implements remote CORBA objects, programmers must be aware that parameter passing in CORBA operations differs slightly from conventional programming. Since references to data in memory on one machine have no meaning on another, the transmission of such references requires the entire memory block to be transmitted over the network and reconstructed in the receiver’s memory. So a programmer must take care to avoid memory leakage for both original and new copies of the data.

This section explains the memory management rules that clients and servers must observe in order to ensure that there is no memory leakage on their machines.

**in parameters**

An in parameter is passed from the client to the server. No special memory management rules apply.

On the client side, for all parameter types, the client must pass the BAAN C symbol that represents the data to be sent to the server. Neither CORBA nor the server changes the data; consequently the client is responsible for memory management.

On the server side, an in parameter maps to a BAAN C const parameter that is made available to the server for the duration of the function call. The server is not allowed to change the data; if it does, the changes are not propagated back to the client. The server must make a copy of the data if it wishes to retain it after the call has completed.

**inout parameters**

In the case of inout parameters, a value is passed from the client to the server and vice versa. There are some special rules for string parameters, but not for basic types.

When a client passes a string to a server (or any other type of no fixed length), it must declare the parameter as a based symbol and allocate memory for it. On return, the symbol points to the output value from the server. This can differ in size from the original string.
The symbol now points to a different memory location from that allocated by the client. The client is responsible for releasing this memory using free.mem(). Note that any based symbols that were based on the original symbol are no longer valid.

On the server side, an inout parameter maps to a BAAN C ref parameter. The ref parameter holds the data passed by the client and the server is allowed to modify that data. The server can reallocate the memory for the ref parameter using alloc.mem().

**Note for BAAN IVc users**
The rules for handling inout parameters are different for BAAN IVc and for later versions. The following are the rules for string parameters in BAAN IVc.

On the client side, the symbol passed for an inout string must be long enough to receive the results from the server. A server never returns a string longer than that sent to it by the client. So, if the client declares a string of maximum length ten and then initializes it with a string of only five characters, the server will never return more than five characters. The client must either declare the string as fixed or initialize it with ten characters. For example:

<table>
<thead>
<tr>
<th>BAAN C (use fixed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>string s(10) fixed</td>
</tr>
<tr>
<td>CORBA.Request.invoke ( ..., s, ... )</td>
</tr>
<tr>
<td>BAAN C (initialize fully)</td>
</tr>
<tr>
<td>string t(10)</td>
</tr>
<tr>
<td>t = “0123456789”</td>
</tr>
<tr>
<td>CORBA.Request.invoke ( ..., t, ... )</td>
</tr>
</tbody>
</table>

**out parameters**

An out parameter is passed from the server back to the client. For strings, some memory management rules apply.

On the client side, for all basic types, the client passes the name of a symbol of the appropriate type. On return, this symbol holds the value sent by the server.

When passing types that do not have a fixed length (a string, for example), the client must declare a based symbol and pass it to the server. There is no need to allocate memory first. If the client does so, the ORB will reallocate the memory. On return, the symbol holds the data sent by the server. The ORB allocates the memory required to hold this data; the client must explicitly free the memory when it is no longer needed.

In the following example, a client calls a remote server to request the full name of a person named Jan:

<table>
<thead>
<tr>
<th>BAAN C (use fixed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>string s(10) fixed</td>
</tr>
<tr>
<td>CORBA.Request.invoke ( ..., s, ... )</td>
</tr>
<tr>
<td>BAAN C (initialize fully)</td>
</tr>
<tr>
<td>string t(10)</td>
</tr>
<tr>
<td>t = “0123456789”</td>
</tr>
<tr>
<td>CORBA.Request.invoke ( ..., t, ... )</td>
</tr>
</tbody>
</table>
// IDL
interface ttstpnames {
    void expandName ( in string name, out string result );
};

| BAAN C
string fullName based
CORBA.Request.invoke( ..., “expandName”, “Jan”, fullname, ...
) message ( “Jan is ” & fullName )
free.mem ( fullName )

The call to free.mem() is necessary to avoid memory leakage.

Note that it is a special ORB feature that a based string is allocated in another function. The same is not possible in application-defined functions.

On the server side, an out parameter maps to a BAAN C ref parameter. For all basic types, the server can put a value in the ref parameter to be returned to the client. The server does not need to allocate memory for basic types.

If the type is a string, then the server must call alloc.mem() to allocate memory for the return value; the ORB frees the memory on return. If the server wants to retain the data, it must make a copy of it.

The following is the implementation of the ttstpnames interface defined in the previous example:

| BAAN C – DLL ‘names’
function extern void ttstpnames.expandName ( const string name(), ref string result ) {
    if name = ”Jan” then
        alloc.mem ( result, 12 )
        result = “Jan Klaassen”
    endif
}
Return result passing considerations

Return values are similar to out parameters. For all types of the return value, the server can return a constant, a locally allocated symbol, or a based symbol. For example:

// IDL
interface ttstpaccount {
    string<10> getName ( in short id );
    string setName ( in short id, in string name );
    short getId ( in string name );
};

| BAAN C |
function extern string ttstpaccount.getName ( const long id ) {
    return ( “Jan Klaassen” )
}
function extern string ttstpaccount.setName ( const long id, const string name() ) {
    string s based
    alloc.mem ( s, 100 )
    s = ... & name & ...
    return ( s )
}
function extern long ttstpaccount.getId ( const string name() ) {
    long id
    id = 10
    return ( id )
}
In the context of ORB Integration, a BAAN server is a BAAN C object that provides services to other applications on the CORBA bus.

Any CORBA-compliant application can connect to a BAAN server and use its services. The client does not need to know how the services are implemented, what programming language they are implemented in, or even where the server is located. The connection to a BAAN server is completely transparent to the client.

All CORBA servers, regardless of their programming language, provide descriptions of their interfaces in IDL. Those descriptions are stored in an Interface Repository. From the IDL, clients obtain all they need to know about the services offered by a particular server: that is, its types, attributes, and operations.

Similarly, a CORBA server does not need to know where the client is located, what language it is written in, or whether requests are generated statically or dynamically. The ORB handles all incoming requests; it finds the object that can implement the request, generates the object reference, passes the parameters of the request to the object, invokes the appropriate operation, and returns the results.

The server and client can run on different machines in the distributed system, in different address spaces in the one machine, or within the same address space on a machine.

This chapter describes how to create a BAAN server. Chapter 5 describes how to create a BAAN client that uses the services of this server.
BAAN DLLs as CORBA objects

In BAAN IV Tools, functions for common use are implemented as Dynamic Link Libraries (DLLs). In BAAN C, CORBA objects are also implemented as DLLs.

The following diagram illustrates how a BAAN DLL maps to a CORBA object.

The services provided by a BAAN server are defined in IDL interfaces. Each IDL interface maps to a separate BAAN DLL with the same name. This DLL implements the operations in the interface.

The DLL code must implement the IDL definition exactly. The ORB then takes care of invoking the appropriate functions in the DLL when a request is received.

The DLL itself resides in a server. For BAAN C objects, the server is always the BAAN bshell.
Creating a BAAN server: overview

Creating a BAAN server consists of the following steps:

1. Define the IDL interfaces.
2. Create the DLL.
3. Register the IDL interfaces in the Interface Repository.

The following sections describe these steps in detail for creating a grid server. The server creates a two-dimensional grid of fixed size. The operations it exposes to client applications, via the IDL, enable clients to obtain the grid size. They also enable clients to address individual grid cells by their row/column coordinates in order to write values to and retrieve values from the cells.
Step 1: defining the IDL interfaces

The first step in creating a BAAN server is to define the IDL interfaces. The interface to the grid server is defined in IDL as follows:

```idl
// IDL testgrid.idl

interface testgrid {
    readonly attribute short height;  // height of the grid
    readonly attribute short width;   // width of the grid

    // IDL operations

    // set the element [n,m] of the grid to value:
    void set(in short n, in short m, in long value);

    // return the element [n,m] of the grid:
    long get(in short n, in short m);
};
```

The interface provides two attributes, `height` and `width`, which define the size of the grid. Since these are labeled `readonly`, they cannot be directly modified.

The interface also defines two operations:
- `set()`, which is used to write a value to a specified cell in the grid.
- `get()`, which is used to read the value in a specified cell.

The parameters to the operations are labeled as `in`, which means that they are being passed from the client to the server.

Further operations could be added, for example, to modify the size of the grid.

The name of the interface maps to the name of the BAAN DLL that implements the interface. Because a BAAN DLL resides in a package and a module, the IDL name is a concatenation of the package, module, and library names.

The first two letters indicate the BAAN package (`te` for the grid server); the next three letters indicate the BAAN module (`tst` for the grid server). The remaining letters indicate the name of the DLL (`grid` for the grid server). To implement the server defined here, a DLL with the name `grid` must be created in the module `tst` in the package `te`. 
Step 2: creating the DLL

The second step in creating a BAAN server is to create the DLL. This must contain a complete implementation of the interface defined in the IDL file, using fully scoped names. Operation and attribute names do not have to be scoped for a BAAN server; the name of the DLL already scopes them. However, for consistency, and to avoid name conflicts when local clients use the DLL objects, fully scoped names are normally used.

Writing the DLL code: attributes

A grid object has two attributes: height and width. The server code begins by defining these attributes, assigning them values, and allocating storage for those values:

```c
#define g_height 99
#define g_width 88
long g_a ( g_height, g_width )
```

Access to these attributes is provided by accessor functions. Because the attributes are defined as `readonly`, there is a single accessor function only for each attribute: that is, a `_get_` function to retrieve the attribute value. Because the functions must be accessible in other programs, they are declared as external functions. Note that an IDL basic type `short` maps to a `long` in BAAN C.

The accessor functions are declared as follows:

```c
function extern long tetstgrid._get_height ( )
function extern long tetstgrid._get_width ( )
```

Writing the DLL code: operations

The two IDL defined operations (`set()` and `get()`) map to two `extern` functions of the same name in BAAN C. Note that the IDL basic type `short` maps to a `long` in BAAN C; and an IDL `in` parameter maps to a `const` parameter in BAAN C.

The functions are declared as follows:

```c
function extern void tetstgrid.set ( const long n, const long m, const long value )
function extern long tetstgrid.get ( const long n, const long m )
```
The server DLL

The full code for the grid DLL is as follows:

| BAAN C |
| tetstgrid.dll |

#define g_height 99
#define g_width 88

private data (the actual storage)
long g_a ( g_height, g_width )

IDL: readonly attribute short height
function extern long tetstgrid._get_height ( )
{
    return( g_height )
}

IDL: readonly attribute short width;
function extern long tetstgrid._get_width ( )
{
    return { g_width }
}

IDL: void set( in short n, in short m, in long value);
function extern void tetstgrid.set ( const long n, const long m,
const long value )
{
    g_a ( n, m ) = value
}

IDL: long get( in short n, in short m )
function extern long tetstgrid.get ( const long n, const long m )
{
    return { g_a ( n, m ) }
}
Registering the IDL interfaces

The third step in creating a BAAN server is to register the IDL interfaces in the Interface Repository so that client applications and the ORB can access this information.

To register the grid server, use the following command on the server’s host:

```
idl -R testgrid.idl
```
In the context of ORB integration, a BAAN client is a BAAN C application that uses the services of a CORBA server.

A BAAN client can connect to any server on the CORBA bus, regardless of how the services it is accessing are implemented, the programming language used, or the location of the server. The connection to a server is completely transparent to the client.

The server and client can run on different machines in the distributed system, or in different address spaces in the one machine. Also, the term client is relative: the same application can be both a client and a server at the same time. It is a client when using the services of other CORBA objects; it is a server when other CORBA objects are using services it provides.

This chapter describes how to create a BAAN client. Chapter 4 describes how to create the BAAN server that provides the services the client is using. Chapter 6 provides full details of the BAAN C client API.

**Static and dynamic method invocation**

A client uses the services of a CORBA object by invoking the object’s methods (operations). CORBA defines two invocation interfaces: Static Invocation Interface (SII) and Dynamic Invocation Interface (DII). In both cases, the client performs a request by specifying the object reference (or object id), the operation that performs the required service, and the parameters of the operation. The invocation interface used is completely transparent to the server.

The SII consists of client IDL stubs that are generated by an IDL compiler. To make a request, the client calls a stub routine that is specific to a particular operation on a particular object. From the programmer’s perspective, a call to a remote object uses the same invocation syntax as a call to a local object; there is also direct mapping between the IDL interface and the function call. No IDL compiler is available with the current version of BAAN ORB Integration.

The SII provides a simple and natural interface to remote objects. However, there are disadvantages. For example, the IDL interfaces that a client program can use are limited to those selected by the programmer when building the client application. Also, deferred synchronous calls are not supported by the SII.
The DII is more complex but also more flexible. It provides a generic way to call a server, and allows an application to issue requests for any interface, even if that interface was unknown at the time the application was compiled. This means that the IDL interfaces used by a client do not have to be statically determined at the time the program is designed and implemented. It also means that there is no direct mapping between an IDL interface and the client program.

Instead of calling a stub routine that is specific to a particular operation on a particular object, a client program can invoke a remote object by specifying, at run time, the target object reference, the operation/attribute name, and the parameters to be passed.

BAAN C client applications use the DII to access remote objects. The mapping to the DII for BAAN C provides facilities for:

- Obtaining an object reference.
- Issuing calls on the object reference.
- Releasing the object reference.

The following diagram illustrates how client and server code maps to the IDL. As already discussed, there is no direct mapping from IDL to BAAN clients; instead, the client maps to the DII.

```
IDL
interface example {
  long operation(param, param)
}

BAAN server (DLL)
function extern
long example.operation(param, param)
{
  . . .
}

BAAN client (DII)
CORBA.Request.invoke
(id, "example", "operation", retval, param, param)
```

BAAN ORB Integration Programmer’s Manual
5-2
Writing the BAAN client code: overview

The steps involved in accessing and using a remote object are as follows:

1. Obtain object reference
   - `string_to_object()`
   - Create proxy and return object reference

2. Create request and invoke method
   - `invoke(object reference, method, arguments)`
     - or
   - `send(object reference, method, arguments)`
   - Get response

3. Release object
   - `release()`

1. Obtain an object reference.
2. Create and invoke a request.
3. Release the object.

The following sections describe the first three steps in detail for writing a client that accesses the grid server created using the procedures described in Chapter 4, Creating a BAAN server. The server creates a two-dimensional grid of fixed size. The operations it exposes to client applications enable clients to obtain the grid size and to address individual cells in the grid by their row/column coordinates in order to write values to those cells and to retrieve values from the cells.

Once the client application has been written, it is compiled and run in the same way as any other BAAN application.
Step 1: obtaining the object reference

The first step in writing a BAAN C client program that accesses the grid server object is to obtain an object reference for a grid object. You do this using the CORBA.ORB.string_to_object() function, which is part of the DII (see Chapter 6, BAAN client API, for full details):

```c
long id
id = CORBA.ORB.string_to_object("abe", "bshell", ",", "tetstgrid" )
```

The function parameters specify the names of the target host, server, and DLL. The function creates a proxy object for the remote grid object and returns an object reference for the proxy object.

The last parameter (tetstgrid), is the interface name as specified in the IDL code. For BAAN servers, this is also the name of the DLL. Because a BAAN DLL resides in a package and a module, the IDL name is a concatenation of the package, module, and library names.

The first two letters indicate the BAAN package (te for the grid server); the next three letters indicate the BAAN module (tst for the grid server). The remaining letters indicate the name of the DLL (grid for the grid server).

Step 2: creating and invoking a request

The second step in accessing the grid object is to create and invoke a request on it to perform some operation.

In some programming languages, multiple statements are required to do this. In BAAN C, a single function creates the request and invokes the required operation. There are two options: CORBA.Request.invoke() which is used for synchronous object calls and CORBA.Request.send() which is used for deferred synchronous object calls (see Chapter 6, BAAN client API).

Accessing the grid attributes

The IDL interface for the grid object provides two attributes, height and width, which define the size of the grid. In general, access to an attribute is provided by two accessor functions, one to set the value of the attribute (_set_<attribute>) and one to retrieve the value of the attribute (_get_<attribute>). Because height and width are readonly attributes, the grid server provides only a single accessor function for each (that is, _get_height and _get_width).
To determine the height and width of the grid object, the client application makes the following calls (for clarity, error handling is omitted):

```java
long height, width

CORBA.Request.invoke(id, "_get_height", height )
message("Height = %d", height)

CORBA.Request.invoke(id, "_get_width", width )
message("Width = %d", width)
```

Note that the return value is the first variable parameter of the function.

### Setting and retrieving cell values

The IDL interface for the grid object provides two operations for setting and retrieving the value in a particular cell in the grid. To use these operations, the client application makes the following calls:

```java
long height, width, value

CORBA.Request.invoke(id, "set", 2, 4, 123 )
CORBA.Request.invoke(id, "get", value, 2, 4 )
```

The parameters to the first `invoke()` function are the object reference, the name of the operation, the row/column coordinates for the target cell, and the value to be entered in the cell. The parameters to the second `invoke()` function are the object reference, the name of the operation, the return value, and the row/column coordinates for the target cell.

Again, note that when there is a return value, it is returned in the first variable parameter. When there is no return value, the first variable parameter is the first parameter of the operation.

### Step 3: Releasing the grid object

The final step in using the grid object is to release it when it is no longer required. You use `CORBA.ORB.Release()` to do this (see Chapter 6, BAAN client API, for full details). This function deletes the proxy object and releases the remote object. It is part of the DII.

To release the grid object, the client application makes the following call:

```java
CORBA.ORB.Release(id )
```
The client program

The full code for the grid client application is as follows (for clarity, error handling is omitted):

```plaintext
function main()
{
    long id
    long height, width, value

    id = CORBA.ORB.string_to_object("abe", "bshell", "", "testgrid")

    CORBA.Request.invoke(id, "_get_height", height)
    message("Height = %d", height)

    CORBA.Request.invoke(id, "_get_width", width)
    message("Width = %d", width)

    CORBA.Request.invoke(id, "set", 2, 4, 123)
    CORBA.Request.invoke(id, "get", value, 2, 4)

    if value != 123 then
        message("Grid failed!"
    endif

    CORBA.ORB.Release(id)
}
```
6 BAAN client API

The BAAN C application programming interface (API) provides a set of functions that enable BAAN client applications to access remote objects. Any remote object for which an IDL interface has been defined can be accessed using the API, provided that the object’s server is registered with Orbix.

Interfaces

The CORBA extension to BAAN C consists of two interfaces: Request and ORB. These implement the Request and ORB interfaces defined in the CORBA specification.

The ORB interface provides functions for binding to and releasing a remote object.

The Request interface provides functions for sending requests to a remote object and for retrieving responses from the object. It is part of the Dynamic Invocation Interface (DII); this is the CORBA-defined interface that BAAN C clients use to invoke remote objects (see Chapters 1 and 5 for further details).

CORBA module

All CORBA-defined names are treated as if they were defined within a module named CORBA and must be referred to by their fully scoped names. This is done to prevent names defined in the CORBA specification from clashing with names in BAAN C and BAAN libraries. For example, the invoke operation in the Request interface must be referred to as CORBA.Request.invoke().
Functions

The BAAN C API provides the functions listed in the following table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA.available</td>
<td>Check the availability of CORBA support</td>
</tr>
<tr>
<td>CORBA.ORB.string_to_object</td>
<td>Create a proxy object</td>
</tr>
<tr>
<td>CORBA.ORB.Release</td>
<td>Release an object</td>
</tr>
<tr>
<td>CORBA.Request.invoke</td>
<td>Synchronous request on an object</td>
</tr>
<tr>
<td>CORBA.Request.send</td>
<td>Deferred synchronous request on an object</td>
</tr>
<tr>
<td>CORBA.Request.get_response</td>
<td>Retrieve response on a request</td>
</tr>
<tr>
<td>CORBA.Request.object</td>
<td>Retrieve the object id associated with a particular request</td>
</tr>
<tr>
<td>CORBA.Request.delete</td>
<td>Delete a request</td>
</tr>
</tbody>
</table>

Determining CORBA availability

CORBA.available() is normally the first call in a program that accesses remote objects. It checks if CORBA support is available on the current BAAN platform.

Binding to and releasing an object

CORBA.ORB.string_to_object() is used to create a proxy object for a remote object. It returns an object reference (or id) that other functions subsequently use to identify the object. CORBA.ORB.Release() is used to release the remote object when the program has finished with it.

Making requests

CORBA.Request.invoke() and CORBA.Request.send() are used to make requests on a remote object; the requests specify a target object, an operation to be performed, and the parameters for the operation.

The invoke() function creates, invokes, and subsequently deletes a synchronous request. Normally, the caller is blocked until the target object has processed the request and the function returns with the reply. However, if the operation being invoked is defined as oneway in its IDL definition, invoke() does not wait while the request is being processed. Consequently the caller is not blocked.
The `send()` function creates and send a deferred synchronous request. This is a nonblocking request; the caller may continue in parallel while the server is processing the call. The caller retrieves the response to the request using `CORBA.Request.get_response()`. The `send()` function does not automatically delete the request once it has been processed and the reply retrieved; the caller must use `CORBA.Request.delete()` to do this.

**CORBA.available()**

**Syntax**

```c
long CORBA.available()
```

**Description**

Use this function to check if CORBA support is available on the current BAAN platform. Normally, this is the first call in a program that accesses remote objects.

**Return value**

The function returns TRUE if CORBA support is available on the current BAAN platform (that is, if the CORBA functions are implemented in the bshell). It returns FALSE if CORBA is not supported. In the latter case, the other API functions are still available but they do not do anything.

**CORBA.ORB.string_to_object()**

**Syntax**

```c
long CORBA.ORB.string_to_object(const string host, const string server, const string marker, const string interface)
```

**Description**

Use this function to get access to a remote object. The function creates a proxy object and returns an object reference (object id) for it. When finished with the object, use `CORBA.ORB.Release()` to release it (see Mapping of function parameters to ORB elements: overview, at the end of this chapter).
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>The host name of the target object (for example, baan.utrecht.nl). To bind to an object at a specific location, set this to the relevant host name. If you pass an empty string (&quot;&quot;), the Locator is used to find the host on which the specified server is located (see Chapter 2, BAAN ORB Integration: description and architecture, for further information on the Locator).</td>
</tr>
<tr>
<td>server</td>
<td>The name of the target object's server. For a BAAN server, this is the name of the BAAN bshell (normally, bshell).</td>
</tr>
<tr>
<td>marker</td>
<td>Reserved parameter. This must be specified as an empty string (&quot;&quot;&quot;).</td>
</tr>
<tr>
<td>interface</td>
<td>The name of the target object's interface. When the server is a BAAN bshell, this is the name of the DLL.</td>
</tr>
</tbody>
</table>

Return value

When successful, this function returns an object reference for the proxy object created; otherwise it returns a negative value.

**CORBA.ORB.Release()**

Syntax

```c
long CORBA.ORB.Release(const long id)
```

Description

This function releases the object with the specified id. The id is the object reference, as returned by CORBA.ORB.string_to_object().

When a program has finished with an object, it must call this function in order to delete the proxy object and release the remote object.

Return value

This function returns 0 when successful, or a negative value if an error occurred.
CORBA.Request.invoke()

Syntax

long CORBA.Request.invoke(const long id, const string method
[, ref retval] [, [ref] arg, ...])

Description

This function is used for synchronous object calls. It makes a blocking request on an object; that is, control returns to the caller only when the specified operation has been processed or an error occurs. The function performs the following steps:

- A request is built with the specified input parameters.
- The request is sent to the remote object.
- The function waits until a response is available.
- When an operation has a return value or out/inout parameters, the result values are passed back into the CORBA.Request.invoke() parameters when the operation has completed.
- CORBA.Request.invoke() returns.

If the operation invoked by CORBA.Request.invoke() is defined as oneway in its IDL definition, then the caller is obviously not blocked while the server is processing the call. In this case, the function performs the following steps:

- A request is built with the specified input parameters.
- The request is sent to the remote object.
- CORBA.Request.invoke() returns immediately.

Because oneway operations do not return, you cannot use output parameters or an operation return value when calling CORBA.Request.invoke() in oneway mode.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id</code></td>
<td>The object reference for the target object, as returned by <code>CORBA.ORB.string_to_object()</code>.</td>
</tr>
<tr>
<td><code>method</code></td>
<td>The name of the operation (or method) to invoke. Do not use fully-scoped names here.</td>
</tr>
<tr>
<td><code>retval</code></td>
<td>The parameters following <code>method</code> are variable. If the operation has a return value, it is returned in the first variable parameter (<code>retval</code>). The succeeding parameters are the parameters of the operation. If the operation has no return value, the first variable parameter is the first parameter of the operation.</td>
</tr>
</tbody>
</table>

Return value

This function returns 0 when successful, or a negative value if an error occurred.

Example (two-way)

```c
// plan.idl
interface project {
    string ProjectName();
    void SetPeriod( in string Name, in long period );
    void ReCalculate();
};

BAAN C
LONG id, ret
STRING name(80)
IF NOT CORBA.available() THEN
    message("FATAL ERROR: No support for CORBA")
    abort()
ENDIF
id = CORBA.ORB.string_to_object("orbix.baan.nl", "plan", "", "project")
ret = CORBA.Request.invoke(id, "ProjectName", name)
IF ret < 0 THEN
    message("ERROR: cannot call object. Reason = %d", ret)
ELSE
    message("Project name is : %s", name)
ENDIF
CORBA.Request.invoke(id, "SetPeriod", "BAANVIII", 30)
CORBA.Request.invoke(id, "ReCalculate")
CORBA.ORB.Release(id)
```
Example (one-way)

// IDL
interface administrator {
  oneway void set_date( in string date );
};

| BAAN C
LONG id, ret
id = CORBA.ORB.string_to_object("walker.baan.nl", "bshell", ", " ", 
  "administrator")
ret = CORBA.Request.invoke(id, "set_date", "140297")
IF ret < 0 THEN
  message("ERROR: cannot call object. Reason = %d", ret)
  CORBA.ORB.Release(id)
ENDIF

CORBA.Request.send()

Syntax

long CORBA.Request.send(const long id, const string method 
[, ref retval] [, [ref] arg, ...])

Description

This function is used for deferred synchronous calls on an object. It makes a
nonblocking request on an object; that is, the caller may continue while the
server is processing the call. When a reply is available, a BMS message is sent to
the caller to indicate this. CORBA.Request.get_response() can then be
used to retrieve the results.
The steps involved when using this function are:

- A request is built with the specified input parameters.
- The request is sent to the remote object.
- CORBA.Request.send() returns. The return value is an id for the request (request_id).
- When the reply is available, a BMS message is sent to the caller to indicate this. The message contains the request_id for the particular request to which the reply relates. Note that the message uses the mask CORBA.reply to indicate that it relates to a reply to a CORBA request; this mask must therefore be set on in the calling program.

The calling program can now use the function evt.bms.command() to retrieve the request_id, and can then use CORBA.Request.get_response() to retrieve the reply. Once the reply has been retrieved, the calling program must delete the request using CORBA.Request.delete().

**Parameters**

<table>
<thead>
<tr>
<th>id</th>
<th>The object reference for the target object, as returned by CORBA.ORB.string_to_object().</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>The name of the operation (or method) to invoke. Do not use fully-scoped names here.</td>
</tr>
<tr>
<td>retval, arg</td>
<td>The parameters following method are variable. If the operation has a return value, this is the first variable parameter (retval). The succeeding parameters are the parameters of the operation. If the operation has no return value, the first variable parameter is the first parameter of the operation. Because CORBA.Request.send() returns before the reply is available, the results of an operation are not passed back via the function parameters. Only the in and inout parameters are used; however, for consistency, output parameters are included.</td>
</tr>
</tbody>
</table>

**Return value**

When successful, this function returns an id for the request (request_id). Otherwise it returns a negative value.
Example

// file.idl
interface file {
   long status( out string S, out string S2);
};

| BAAN C
STRING s(80), s2(80)
STRING mask(80)
STRING msg(80)
LONG id, ret, i, request_id, result
LONG event(EVTMAXSIZE)
LONG nbytes, level, evttyple
LONG tim.id

| When using CORBA.Request.send() to make deferred synchronous calls, the session receives a message with mask "CORBA.reply" when the reply is available. So set that mask.
| bms.add.mask("CORBA.reply")

| Create a proxy for the object 'file' in the Server 'file'.
| id = CORBA.ORB.string_to_object("orbix.baan.nl", "file", "", "file")
FOR i = 1 TO 5
   | Send a request
   | Note: With CORBA.Request.send() you cannot omit variable output parameters, even though they are not used to receive data.
   | request_id = CORBA.Request.send(id, "status", result, s, s2)
   message("Request with id %d is pending.", request_id)
   | Wait for the message that is sent when the reply is available.
   | Use a timeout of 2000 milliseconds
REPEAT
   tim.id = set.timer(2000)
   next.event(event)
kill.timer(tim.id)
ret = bms.receive.buffer(msg,nbytes,level,0,mask)
evtttype = evt.type(event)
UNTIL mask = "CORBA.reply" OR evtttype = EVTTIMEREVENT
IF evtttype = EVTTIMEREVENT THEN
  message("Timeout on waiting for reply")
ELSE
  request_id = evt.bms.command(event)
  message("Reply on request #d of object #d", request_id, CORBA.Request.object(request_id))
  ret = CORBA.Request.get_response(request_id, result, s, s2)
  IF ret = 0 THEN
    message("Result: %d, Status: %s/%s", result, s, s2)
  ELSE
    message("No status. Reason = %d", ret)
  ENDFENDIF
  CORBA.Request.delete(request_id)
ENDIF
ENDFOR
ret = CORBA.ORB.Release(id)

**CORBA.Request.get_response()**

**Syntax**

long CORBA.Request.get_response(const long request_id [, ref retval] [, [ref] arg, ...])

**Description**

Use this function to retrieve the results of a request sent using CORBA.Request.send().

When the reply for a request is available, the caller receives a BMS message to indicate this. The message contains the request id for the particular request to which the reply relates. The caller can then specify this request_id in CORBA.Request.get_response() to retrieve the return value and output parameters.

Once the reply has been retrieved, the calling program must delete the request using CORBA.Request.delete().
See CORBA.Request.send() for an example of using this function.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_id</td>
<td>The id for the request whose results are being retrieved, as returned by CORBA.Request.send().</td>
</tr>
<tr>
<td>retval arg ...</td>
<td>These contain the variable parameter list. If the operation has a return value, this is the first variable parameter (retval). The succeeding parameters are the parameters of the operation. If the operation has no return value, the first variable parameter is the first parameter of the operation. For consistency, the variable parameter lists in CORBA.Request.send() and CORBA.Request.get_response() must be identical. Accordingly, CORBA.Request.get_response() must include the operation input parameters as well as its output parameters and return value.</td>
</tr>
</tbody>
</table>

Return value

This function returns 0 when successful, or a negative value if an error occurred.

CORBA.Request.object

Syntax

```
long CORBA.Request.object(const long request_id)
```

Description

Each request made using CORBA.Request.send() is related to a particular remote object. You use CORBA.Request.object() to retrieve the id of the object with which a particular request is associated.

The input parameter is the request_id, as returned by CORBA.Request.send(). The return value is the object id, as returned by CORBA.ORB.string_to_object().

See CORBA.Request.send() for an example of using this function.
Return value

When successful, this function returns the id of the object with which the specified request is associated. It returns a negative value if an error occurred.

**CORBA.Request.delete**

**Syntax**

```c
long CORBA.Request.delete(const long request_id)
```

**Description**

Use this function to delete a request sent using `CORBA.Request.send()`. The final step of a `send()` sequence should always be a call to this function.

**Return value**

This function returns 0 when successful, or a negative value if an error occurred.
Mapping of function parameters to ORB elements: overview

The following diagram illustrates how BAAN C API function parameters map to ORB elements.

```c
id = CORBA.ORB.string_to_object(
    "orbix", "bshell", "", "grid";

CORBA.Request.invoke(
    id,
    "get",
    value,
    2,
    4);
```
## Error codes

Some functions return negative values when an error occurs. The meanings of these values are indicated in the following table:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Value</th>
<th>Problem/Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC_SUCCESS</td>
<td>0</td>
<td>No error.</td>
</tr>
<tr>
<td>DC_INTERNAL_ERROR</td>
<td>-1</td>
<td>An error occurred in the code of a BAAN C function.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>This is a serious error. Report it to Baan.</em></td>
</tr>
<tr>
<td>DC_INVALID_PARAMETER</td>
<td>-2</td>
<td>One or more of the parameter(s) is/are not valid.</td>
</tr>
<tr>
<td>DC_OBJECT_NOT_FOUND</td>
<td>-3</td>
<td>The remote object is not present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Run Orbixd and register the object's server. When using the Locator, check the Locator list.</em></td>
</tr>
<tr>
<td>DC_ERROR_IN_IR</td>
<td>-4</td>
<td>The interface cannot be found in the Interface Repository.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Run Orbixd and register IR. Add interface via idl –R command.</em></td>
</tr>
<tr>
<td>DC_MAXIMUM_REACHED</td>
<td>-5</td>
<td>The maximum number of objects that can be bound at the same time has been reached.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Bind to fewer objects simultaneously.</em></td>
</tr>
<tr>
<td>DC_MEM_ALLOC_FAILURE</td>
<td>-6</td>
<td>No more memory is available.</td>
</tr>
<tr>
<td>DC_REQUEST_FAILED</td>
<td>-7</td>
<td>Cannot create a request.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Make the remote object available.</em></td>
</tr>
<tr>
<td>DC_INVOKE_FAILED</td>
<td>-8</td>
<td>Cannot invoke the remote object.</td>
</tr>
</tbody>
</table>
This chapter describes the procedures for installing and configuring the ORB Integration 1.0 Development Kit for BAAN IVc.

**UNIX**

To install and configure the ORB Integration 1.0 Development Kit on UNIX platforms:

1. Install the BAAN ORB Integration files on the host machine.
2. Add ORB-specific statements to `rc.start`.
3. Add ORB-specific statements to `rc.stop`.
4. In `ipc_info`, add a line that maps the BAAN server’s logical name to the name of the corresponding bshell executable file.
5. Create a new user account for Orbix.

**Step 1**

The following table lists the files that you must install on the host machine and the directories where you must store them. The table also indicates the files on which you must run `chmod +x`.

<table>
<thead>
<tr>
<th>Copy</th>
<th>To</th>
<th>Run chmod +x</th>
</tr>
</thead>
<tbody>
<tr>
<td>orb_srv6.1</td>
<td>$BSE/bin</td>
<td>yes</td>
</tr>
<tr>
<td>orb_srv</td>
<td>$BSE/bin</td>
<td>yes</td>
</tr>
<tr>
<td>bic_global</td>
<td>$BSE/include6.1</td>
<td>no</td>
</tr>
<tr>
<td>rc.start_orb</td>
<td>$BSE/etc</td>
<td>yes</td>
</tr>
<tr>
<td>rc.start_ir</td>
<td>$BSE/etc</td>
<td>yes</td>
</tr>
<tr>
<td>rc.start_deamon</td>
<td>$BSE/etc</td>
<td>yes</td>
</tr>
<tr>
<td>bshell6.1_orb</td>
<td>$BSE/bin</td>
<td>yes</td>
</tr>
<tr>
<td>omidorb_srv</td>
<td>$BSE/tools/ttB40_a/ottmid</td>
<td>no</td>
</tr>
</tbody>
</table>
Step 2
Add the following statements to $BSE/etc/rc.start:

```
set ORB=<name of Orbix directory; this is usually /opt/Orbix_2.1>
set SRV=<name of BAAN CORBA server; for example, bshell4c>
export ORB
export SRV
$BSE/etc/rc.start_orb -start
```

Step 3
Add the following statements to $BSE/etc/rc.stop:

```
set ORB=<name of Orbix directory; this is usually /opt/Orbix_2.1>
set SRV=<name of BAAN CORBA server; for example, bshell4c>
export ORB
export SRV
$BSE/etc/rc.start_orb -stop
```

Step 4
Add the following line to $BSE/lib/ipc_info:

```
<name of CORBA server> s 0 0 p ${BSE}/bin/bshell6.1_orb
```
You must set <name of the CORBA server> to the same value as the SRV variable in rc.start and rc.stop.

Step 5
Add a new user, orbixusr, to the primary group bsp.

SRV variable: overview
A host machine can hold more than one BAAN installation. Each distinct installation is referred to as a BAAN environment. You must separately install and configure each BAAN environment using the steps outlined above.

There is only a single Orbix installation on each host. The startup of Orbix and BAAN are integrated in such a way that you can start and run multiple BAAN environments simultaneously although only a single Orbix environment is started.
In order for client applications to connect to a particular BAAN environment, that BAAN environment must have a unique name. You set the name for each environment by using the SRV variable (see Steps 2 and 3 above). This is the name under which the server is registered with Orbix. It is also the server name specified by a client when connecting to a BAAN CORBA object by using CORBA.ORB.string_to_object().

For example, say you have a host on which two BAAN environments have been installed. One environment has the SRV variable set to bshell4c, the other has the SRV variable set to bshell4c_test. These are the logical names for two distinct ORB-enabled bshells (or servers) in two distinct environments.

Clients can connect to either server, provided that they specify the server’s unique name as indicated in the SRV variable.

Both Orbix and BAAN use a similar mapping mechanism to map the logical name of a server to actual executable files. In Orbix, the mapping is contained in the Implementation Repository. In BAAN, the mapping is contained in the file $BSE/lib/ipc_info.

The following are sample mappings for the server name bshell4c:

- Implementation Repository
  - bshell4c maps to /data/baan4c/bse/bin/orb_srv
  - $BSE/lib/ipc_info
    - bshell4c maps to /data/baan4c/bse/bin/bshell6.1_orb

When the client initiates a connection to the bshell4c server, the following events occur:

- Orbix consults the Implementation Repository to find the name of the executable file that corresponds to the logical server name bshell4c.
- Orbix then launches orb_srv. This is a program that initiates the execution of a bshell named bshell4c.
- BAAN consults the ipc_info file to find the name of the bshell executable that corresponds to the logical name bshell4c.
- BAAN then starts bshell6.1_orb to process the client’s request.
BAAN Windows (BW) configuration

To run a CORBA-enabled BAAN client application from BW, you must configure BW so that the value in its Bshell name field is the same as the value of the SRV variable for the BAAN server that the application will be accessing. To do this:

1. Choose the BW Configuration option.
2. Enter the logical name for the server in the Bshell name field (bshell4c, for example).

Windows NT

To install and configure the ORB Integration 1.0 Development Kit on Windows NT platforms:

1. Install the BAAN ORB Integration files on the host machine.
2. Register the BAAN ORB Integration servers.

Step 1

The following table lists the files that are installed on the host machine and the directories where they are stored.

<table>
<thead>
<tr>
<th>File</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbsrv.exe</td>
<td>\bin</td>
</tr>
<tr>
<td>orbsrv.bat</td>
<td>\bin</td>
</tr>
<tr>
<td>inst_orb.bat</td>
<td>\integration</td>
</tr>
<tr>
<td>ntbshelloorb.exe</td>
<td>\bin</td>
</tr>
<tr>
<td>omidorb_srv</td>
<td>\integration</td>
</tr>
<tr>
<td></td>
<td>(you must copy this file to: \tools\ttB40_*\ottmid)</td>
</tr>
</tbody>
</table>

Note that you must create a directory named ottmid in the Tools VRC, and then copy the file omidorb_srv to that directory.

Step 2

Make sure that orbixd is running on the host. Then execute the inst_orb.bat file, with the BAAN home directory as the first and only argument. This batch file registers the BAAN ORB Integration servers and sets the appropriate privileges for the IR and orbsrv.
BAAN Windows (BW) configuration

To run a CORBA-enabled BAAN client application from BW, you must configure BW so that the value in its Bshell name field is `bshellorb`. To do this:

1. Choose the BW Configuration option.
2. Enter the logical name for the server (`bshellorb`) in the Bshell name field.
This appendix provides some examples of IDL to BAAN C mapping, as it applies to BAAN C clients.

**Supported data types**

BAAN C clients support the following IDL data types:

- Basic types: short, long, unsigned short, unsigned long, float, double, char, boolean, octet
- Bounded and unbounded strings
- Enums
- Constants
- Object references (without interface inheritance)
- Arrays of long and double (for parameters and return values, but not for attributes)

BAAN C server objects support the following IDL data types:

- Basic types: short, long, unsigned short, unsigned long, float, double, char, boolean, octet
- Bounded and unbounded strings
- Enums
- Constants

**Attributes**

The following IDL example defines object attributes of various types, including IDL basic types, enums, and bounded and unbounded strings. The BAAN C code shows the BAAN C client declarations for these attributes and the function calls used to set and retrieve the attribute values. The final attribute is a `readonly` attribute, so there is no function call for setting the attribute value.
// IDL definitions

enum color { red, green, blue, yellow, white };

typedef unsigned long ulong;

attribute float a_float;
attribute long a_long;
attribute ulong a_ulong;
attribute char a_char;
attribute octet a_octet;
attribute string a_string;
attribute string<64> a_string64;
attribute color a_color;
readonly attribute string ra_string;

<table>
<thead>
<tr>
<th>BAAN C client declarations</th>
</tr>
</thead>
</table>
#define color.red 0
#define color.green 1
#define color.blue 2
#define color.yellow 3
#define color.white 4

long object.example
long ret
string tmpstr (1024)
double a_float
long a_long
string a_char (1)
long a_octet
string a_string (1000)
string a_string64 (64)
long a_color
string ra_string (1000)

| BAAN C client: setting attribute values |
ret = CORBA.Request.invoke(object.example, "_set_a_float", 1.24)
ret = CORBA.Request.invoke(object.example, "_set_a_long", 1997)
ret = CORBA.Request.invoke(object.example, "_set_a_char", "f")
ret = CORBA.Request.invoke(object.example, "_set_a_octet", 100)
tmpstr = "Hello world"
ret = CORBA.Request.invoke(object.example, "_set_a_string", tmpstr)
ret = CORBA.Request.invoke(object.example, "_set_a_string64", tmpstr)
ret = CORBA.Request.invoke(object.example, "_set_a_color", color.blue)

| BAAN C client: retrieving attribute values |
ret = CORBA.Request.invoke(object.example, "_get_a_float", a_float)
ret = CORBA.Request.invoke(object.example, "_get_a_long", a_long)
ret = CORBA.Request.invoke(object.example, "_get_a_char", a_char)
ret = CORBA.Request.invoke(object.example, "_get_a_octet", a_octet)
ret = CORBA.Request.invoke(object.example, "_get_a_string", a_string)
ret = CORBA.Request.invoke(object.example, "_get_a_string64", a_string64)
ret = CORBA.Request.invoke(object.example, "_get_a_color", a_color)
ret = CORBA.Request.invoke(object.example, "_get_ra_string", ra_string)

Operations

The following IDL examples illustrate the definition of IDL operations. The BAAN C code shows the relevant BAAN C client declarations and the function calls used to invoke the operations.

Note that in the examples, operation names take the following form:
ret_op_mode_type[_mode_type ...]
ret is the return value of the operation. op indicates that the name identifies an operation. mode indicates the directional mode of a parameter. type indicates the data type of a parameter.

Also, parameter names in the IDL definitions take the following form:
mode_type
where mode indicates the directional mode of the parameter and type indicates its data type.
Operations with parameters of simple type

The following IDL example defines a number of operations with parameters and return values of simple types (including short, unsigned short, unsigned long, and double). The BAAN C code shows the required declarations in a BAAN C client and the function calls used to invoke the operations.

// IDL

typedef unsigned long ulong;
typedef unsigned short ushort;

void void_op_in_short (in short in_short);
void void_op_inout_ulong (inout ulong inout_ulong);
ushort ushort_op_out_double (out double out_double);

| BAAN C declarations
long object.example
long ret
long tmp_short
long tmp_ushort
double tmp_double
long tmp_ulong

| BAAN C calls:
ret = CORBA.Request.invoke(object.example, "void_op_in_short",
  tmp_short)
ret = CORBA.Request.invoke(object.example, "ushort_op_out_double",
  tmp_ushort, tmp_double)
ret = CORBA.Request.invoke(object.example, "void_op_inout_ulong",
  tmp_ulong)
Appendix B: Sample IDL definitions and BAAN C client code

Operations with parameters of type string

The following IDL example defines a number of operations with parameters of type string (bounded and unbounded). The BAAN C code shows the required declarations in a BAAN C client and the function calls used to invoke the operations.

// IDL

typedef string<64> string64;
void void_op_in_string (in string in_string);
long long_op_inout_string (inout string inout_string);
char char_op_out_string (out string out_string);
float float_op_out_string64 (out string64 out_string64);

| BAAN C declarations
long object.example
long ret
string tmp_string (1000)
long tmp_long
string tmp_char (1)
double tmp_float
string tmp_string64 (64)

| BAAN C calls
ret = CORBA.Request.invoke(object.example, "void_op_in_string", "Barneveld")
tmp_string = "Orbix is a Registered Trademark of IONA Technologies Ltd."
ret = CORBA.Request.invoke(object.example, "long_op_inout_string", tmp_long, tmp_string)
ret = CORBA.Request.invoke(object.example, "char_op_out_string", tmp_char, tmp_string)
ret = CORBA.Request.invoke(object.example, "float_op_out_string64", tmp_float, tmp_string64)
One-way operations

The following IDL example defines a oneway operation. The BAAN C code shows the required declaration in a BAAN C client and the function call used to invoke the operation.

// IDL
oneway void oneway_void_op_in_string_in_double
    (in string in_string, in double in_double);

| BAAN C declarations
long object.example
long ret

| BAAN C call
ret = CORBA.Request.invoke(object.example,
    "oneway_void_op_in_string_in_double", "A string", 345.99)

Operations with parameters of type array

The following IDL example defines a number of operations with parameters of type array. The BAAN C code shows the required declarations in a BAAN C client and the function calls used to invoke the operations.

// IDL
typedef long long345 [3] [4] [5];
typedef double double345 [3] [4] [5];

long345 long345_op_in_long345_inout_long345_out_long345
    (in    long345 in_long345,
     inout long345 inout_long345,
     out   long345 out_long345);

double345 double345_op_in_double345_inout_double345_out_double345
    (in    double345 in_double345,
     inout double345 inout_double345,
     out   double345 out_double345);
| BAAN C declarations |

long object.example
long ret
string operation (512)
long r_long345 (3,4,5)
long i_long345 (3,4,5)
long io_long345 (3,4,5)
long o_long345 (3,4,5)
double r_double345 (3,4,5)
double i_double345 (3,4,5)
double io_double345 (3,4,5)
double o_double345 (3,4,5)

| BAAN C calls |

operation = "long345_op_in_long345_inout_long345_out_long345"
ret = CORBA.Request.invoke(object.example, operation, r_long345, i_long345, io_long345, o_long345)

operation = "double345_op_in_double345_inout_double345_out_double345"
ret = CORBA.Request.invoke(object.example, operation, r_double345, i_double345, io_double345, o_double345)
Object references

The following IDL example defines two interfaces: extra and example. The extra interface declares two attributes and a single operation. The example interface provides a single operation that returns a reference for an object that implements the extra interface.

The BAAN C code shows the required declarations in a BAAN C client and some relevant function calls.

// IDL

interface extra
{
    attribute short a_short;
    readonly attribute string<40> a_string40;
    long long_op_in_long_out_string (in long in_long,
                                           out string out_string);
};

interface example
{
    extra extra_op_in_string (in string in_string);
};

| BAAN C declarations
long object.example
long object.extra
long ret
string extra.string40 (40)

| BAAN C calls
operation = "extra_op_in_string"
ret = CORBA.Request.invoke(object.example, operation, object.extra, 
                           "extra object")

operation = ".get_a_string40"
ret = CORBA.Request.invoke(object.extra, operation, extra.string40)
Sample IDL file and server application

The following is an example of a simple banking application for managing bank accounts.

```idl
dll testbank

Implements interface bank in file testbank.idl:

const long max_accounts = 100;
enum status { ok, error, unknown };

interface testbank {
    const float defaultOverdraftLimit = 1000.00;
    enum float { error, overdraft, blocked, ok };
    status makeDeposit ( in long account, in float amount );
    status makeWithdrawal ( in long account, inout float amount );
    float getBalance ( in long account );
    readonly attribute ::status bankStatus;
};

const long max_accounts = 100; (global)
define max_accounts 100

enum status { ok, error, unknown }; (global)
define status.ok 0
define status.error 1
define status.unknown 2

define testbank.defaultOverdraftLimit 1000.00

tenstbank

define testbank.status.error 0
define testbank.status.overdraft 1
define testbank.status.blocked 2
define testbank.status.ok 3

long bankStatus
```
long accountStatus
double accounts ( max_accounts )

function extern long tetstbank.makeDeposit ( const long account,
const double amount ) { 
  if bankStatus = status.ok then
    |<make deposit>
    accountStatus = tetstbank.status.ok
  else
    accountStatus = tetstbank.status.error
  endif
  return ( accountStatus )
}

function extern long tetstbank.makeWithdrawal ( const long account,
const double amount ) {
  if accounts ( account ) + tetstbank.defaultOverdraftLimit >=
  amount then
    |<make withdrawal>
    accountStatus = tetstbank.status.ok
  else
    accountStatus = tetstbank.status.overdraft
  endif
  return ( accountStatus )
}

function extern double tetstbank.getBalance ( const long account ) {
  if account <= max_accounts then
    return ( accounts ( account ) )
  else
    return ( 0.0 )
  endif
}

function extern long tetstbank._get_bankStatus () {
  return ( bankStatus )
}
BAAN ORB Integration maintenance/debugging support is provided by the standard BAAN debug/log system.

Server troubleshooting

When a client connects to a BAAN server and experiences problems, there are a number of log files that can be consulted to help trace the problem.

The log files in the /tmp directory are always available and apply to all users. The log files in the $BSE/log directory are generated by connecting to the debug_bshell instead of the bshell; these files apply to specific users only.

Consult the files in the following order:

- `orbd.log` and `orbd.log.prev` (/tmp directory)
  These contain the output of the Orbix daemon. Consult this to determine whether the daemon received the client’s request for a specific server. This file also shows whether the server, and subsequently the Interface Repository (IR), were successfully started.

- `IR.log` and `IR.log.prev` (/tmp directory)
  These contain the log of the Interface Repository (IR). The IR is used by all BAAN CORBA requests as it contains the information required to translate requests between BAAN and CORBA. Consult this to determine whether the IR started successfully, and whether an error occurred when it found the requested IDL.

- `log.orb_srv.dsi.<name>` ($BSE/log directory)
  This contains a trace of the BAAN ORB Server (specifically, the part that is external to the bshell). Consult this to determine whether the BAAN Server:
  - Started
  - Connected successfully to the bshell and the IR
  - Received the request
  - Returned a reply to the client
Appendix C: Troubleshooting

- **log.orb_srv.ds_link.<name>** ($BSE/log directory)
  
  This logs system events such as an authorization failure when the orb_srv
  starts the bshell.

- **log.orb_srv.3gl.<name>** ($BSE/log directory)
  
  This contains a trace of the BAAN ORB Server (specifically, the part that is a
  bshell process) and of the BAAN C server that was launched by the ORB.
  Consult this to determine whether the bshell received the request from the
  ORB Server external process. This will also show whether the server
  processed the request correctly and returned a reply.

  Note that <name> is the name of the user invoking the BAAN server. Note also
  that the dsi and 3gl log files can be very large. So it is usually better to start
  your trace at the end of the file rather than at the beginning.

  In addition to these ORB log files, there may be useful messages in the standard
  BAAN log files. For example, $BSE/log/log.bshell contains logs of the
  bshell runtime. Consult this, for example, to determine whether the ORB Server
  (internal process) successfully started and whether the requested DLL was found.

**Client troubleshooting**

When a client function returns an error, details of the error are logged in the file
$BSE/log/log.orb_client.<user>.

To trace other problems with a client program, start the bshell with the following
switches:

- dbgorb -keeplog

When using BAAN Windows (BW), choose the BW Configuration option, and
then set the Command field to:

- dbgorb -keeplog

The default logfile for output from this procedure is:

$BSE/tmp/bshell.<pid>

To output to a different file, use the following command:

- dbgorb -keeplog -logfile <filename>

where <filename> is the name of the log file you want to use.
The following limitations apply to BAAN C server and client objects:

- The length of a string is limited to the maximum size currently supported by BAAN C (that is, 16K).
- On the client side, when `alloc.mem()` is used to allocate memory for a based string, the maximum size of the string that can be passed to a remote object is 5 MB.
- A string can consist of all possible 8-bit quantities except the NUL [\0] character.
- BAAN C function can have 255 parameters at the most. This limit applies to CORBA functions also.
- The maximum size of a marshaled request that can be sent to a BAAN C server is 16K. Note that when double quotes ["] are included in a string, these are duplicated in the marshaled request, thus increasing its size.
- Some words are reserved as keywords in BAAN C; others are already defined as functions in the standard program. For example, `set`. Consequently, not every function defined in IDL can be implemented in a BAAN C server.

Since there is no IDL compiler that translates IDL to BAAN C, IDL specifications must be translated manually. The language mapping must be applied precisely and consistently; otherwise errors will be introduced in the BAAN C code. If a runtime error occurs, it may be due to a misspelled name or to type mismatches, for example. This problem can be overcome by generating the IDL code from BAAN DLLs or form definitions; that is, by performing the language mapping in the opposite direction. However, remember that not every DLL can be translated to IDL.