Rational® DOORS

IBM®

API Manual

IBM Rational DOORS

Rational DOORS API Manual

Release 9.2

Before using this information, be sure to read the general information under the "Notices" chapter on page 57.

This edition applies to **IBM Rational DOORS**, **VERSION 9.2**, and to all subsequent releases and modifications until otherwise indicated in new editions.

Table of contents

1 About this manual

Welcome to IBM® Rational® DOORS® 9.2, a powerful tool that helps you to capture, track and manage your user requirements.

This manual describes how to integrate IBM Rational DOORS with other applications. It describes how you can create links between Rational DOORS and external tools; it focuses on the overall strategy for creating tool interfaces.

This manual assumes that you know how to program in C and DXL (DOORS eXtension Language).

Typographical conventions

The following typographical conventions are used in this manual:

Typeface or Symbol	Meaning
Bold	Important items, and items that you can select, including buttons and menus: “Click **Yes** to continue”.
Italics	Book titles.
`Courier`	Commands, files, and directories; computer output: “Edit your `.properties` file”.
>	A menu choice: “Select **File > Open**”. This means select the **File** menu, and then select the **Open** option.

Each function or macro is first introduced by name, followed by a declaration or the syntax, and a short description of the operation it performs. These are supplemented by brief examples where appropriate.

In declarations and syntax, parentheses (( )) are literal language elements, square brackets ([ ]) enclose optional items; braces ({ }) enclose alternatives, which are separated by pipe symbols (|); and ellipsis (...) indicate that arguments can be repeated. Where square brackets or pipe symbols form part of the syntax they are shown in bold.

Terminology

The following terminology is used in this manual:

Term	Description
API	Application Programming Interface. Normally a set of functions and data structure declarations provided by an application program as a means of making its facilities and data available to other programs. In the context of Rational DOORS, DXL can often be used to do the tasks for which other tools would need an object library type interface. For tighter integration, Rational DOORS also supplies a C-based API to create a DXL like layer around the target tool.
Rational DOORS C API	An API written in C which enables a C program to make its own DXL like language or communicate with Rational DOORS using IPC.
DXL	DOORS eXtension Language
IPC	Inter Process Communication. A system of message passing between processes, such as between Rational DOORS and a CASE tool.
TDS	Toy Database Server; An example C-based API provided to illustrate the use of the Rational DOORS C API in linking external tools to Rational DOORS.

Related Documentation

The following table describes where to find information in the documentation set:

For information on	See
What's new in version 9.2 of Rational DOORS	The Rational DOORS readme file
How to install Rational DOORS	*Rational DOORS Installation Guide*
How to set up licenses to use Rational DOORS	*Rational Lifecycle Solutions Licensing Guide*

For information on	**See**
How to use Rational DOORS	*Getting Started with Rational DOORS* *Using Rational DOORS*
How to write requirements	*Get it Right the First Time*
How to set up and manage Rational DOORS	*Managing Rational DOORS*
The DXL programming language	*DXL Reference Manual*
How to integrate Rational DOORS with other applications	*Rational DOORS API Manual*

These documents are on the Rational Information Center at http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/index.jsp.

2 Introduction

This chapter outlines how DXL can be used to link Rational DOORS with external tools. It contains the following topics:

- Rational DOORS APIs
- Rational DOORS and external data
- Strings

Rational DOORS APIs

Rational DOORS provides application programming interfaces (APIs) for extending capability, customizing, and linking to other tools. The main interface is the DOORS eXtension Language (DXL).

DXL provides a comprehensive set of facilities for making links between Rational DOORS and external applications, such as CASE tools or configuration management databases. Links can range from simple file format import or export, through to complex manipulations of externally managed data using interprocess communication (IPC). For example:

- DXL can be used to convert Rational DOORS data into the file format accepted by a user's word processor.
- A two way interactive link can be established between a set of Rational DOORS requirements and their realization in a CASE tool database.

The Rational DOORS C API supports multi-platform tool integrations requiring IPC. It also supports the creation of languages like DXL for the tool being linked to.

For integrations that are to run only on Windows® platforms, DXL supports OLE automation, both as a client and a server application.

File format import or export can be accomplished with a moderate level of programming experience. The DXL server can be used by anyone able to understand simple DXL commands. OLE automation can be used by those with moderate knowledge of DXL and Visual Basic. Complex tool linkage requires both competence in the interfacing facilities provided by the target system and an understanding of the facilities of the Rational DOORS C API.

Rational DOORS and external data

Rational DOORS can read and write several commonly used file formats, for example, FrameMaker and Rich Text Format (RTF). However, it is impossible to anticipate and support every file format that might be used.

Therefore, the facilities Rational DOORS uses for file import and export are available to the user; you access them through DXL.

Importing and exporting files is a task whose complexity depends on the complexity of the input format to be parsed. If you already have a parser, you can extend its capability using the techniques described in "Using the DXL server," on page 17 and "Interactive interfacing with a complex external tool," on page 27.

When developing translation programs, you can use the Rational DOORS source code as a starting point. The code is in:

```
$DOORSHOME/lib/dxl/standard/import
```

and

```
$DOORSHOME/lib/dxl/standard/export
```

Strings

An important aspect of building a successful DXL application, such as an importer, is string handling. Rational DOORS has an internal data structure, called the string table, which stores single copies of ASCII strings used in Rational DOORS. Any string created by a DXL program resides in the string table for the duration of the current Rational DOORS session. You should therefore avoid constructs like:

```
line = line ch ""
```

where `line` is a string being constructed out of individual characters `ch`. This is a very inefficient construct because every temporary value stored in line is made persistent in the string table.

Instead of concatenating characters into a string variable, you should use the `Buffer` data type because buffers do not consume string table space, for example:

```
Buffer Buf
Buf += ch
```

3 The Rational DOORS C API

This chapter describes the Rational DOORS C API. A series of macros and functions allow you to perform integration tasks like those in this manual. Refer also, to the file `$DOORSHOME/include/doors/ api.h`.

This chapter contains the following topics:

- About the API
- Object and library files
- Extending the Rational DOORS C API
- Rational DOORS C API entry points

About the API

The Rational DOORS C API allows you to create a language like DXL around an existing tool. It also provides the inter-process communication facilities needed to establish a link with Rational DOORS.

Object and library files

The following library files are necessary:

```
$DOORSHOME/bin/dxlapi.dll
$DOORSHOME/bin/dxlapi.lib
```

The .lib file is required by the client C application at link time, while the .dll file must be on the path of the client application at run time.

Extending the Rational DOORS C API

This section defines the macros used to extend the Rational DOORS C API interpreter with new functions and data types to create a language like DXL.

New DXL types

When you extend the core DXL language with new operations, you often need new data types which can be passed to the C functions that implement the operations on them. You define new data types using the struct facility, for example:

```
struct Table {}
```

This declaration introduces the new type `Table`.

Introducing new data types is the only valid use of the keyword struct.

apiInstall

Syntax

```
apiInstall(proto,
           fn)
```

Operation

Registers a new function with the API's interpreter. The argument proto is a string containing a valid DXL function prototype, for example:

```
"void create(string)"
```

The argument `fn` is the name of a C function.

The interpreter calls `fn` when the function in proto is executed. The C function `fn` must be declared using `BEGIN_FN`, `END_DECLS`, and `END_FUNCTION`.

When used to install a for loop, proto must be in the form:

```
void ::do(elementType&,
          parentType,
          void)
```

BEGIN_FN

Syntax

```
BEGIN_FN(fn,
         ins,
         outs)
```

Operation

Starts a function declaration. The argument `fn` is the name of the function being declared; it must be the same as the `fn` argument passed to the corresponding call to the `apiInstall` function.

The argument `ins` is the number of input parameters allocated to the DXL function prototype by the corresponding call to `apiInstall`.

The argument `outs` is the return type allocated to the DXL function prototype by the corresponding call to `apiInstall`. The values can be 0 for a void function, or 1 for all other return types.

*P_*

Syntax

```
P_(type,
  var)
```

Operation

Declares a parameter that is accessible with a function declared by `BEGIN_FN` and `END_FN`.

The parameter *`type`* is the type of the parameter. The parameter *`var`* is the variable name of the parameter.

Parameters and variables manipulated by the DXL interpreter must be no larger than a C type long or pointer (whichever is larger). For further information on DXL interpreter data, see “DXL API integration features,” on page 21.

END_DECLS

Syntax

```
END_DECLS
```

Operation

Ends declarations of parameters using `P_`, and other declarations of local variables after a call to `BEGIN_FN`.

*RETURN_*

Syntax

```
RETURN_(value)
```

Operation

Sets the return value after a call to `BEGIN_FN`. The parameter *`value`* is the value to be returned from the function declared using `BEGIN_FN` and `END_FN`.

END_FN

Syntax

```
END_FN
```

Operation

Ends a function declaration started by `BEGIN_FN`.

BEGIN_FOR_DO

Syntax

```
BEGIN_FOR_DO(name,
             pt,
             p,
             et,
             scan)
```

Operation

Starts the declaration of a `for..do` loop, corresponding to the loop installed by the `apiInstall` function.

The argument *`name`* is the name of the loop. The argument *`pt`* is the type of the parent of the loop. The argument *`p`* is a variable that stores the parent. The argument *`et`* is the type of the elements to be scanned. The argument *`scan`* is a variable that holds each scanned element in turn.

PROCESS_DO

Syntax

```
PROCESS_DO(scan)
```

Operation

Continues a `BEGIN_FOR_DO` declaration.

The argument *`scan`* must be the variable passed to `BEGIN_FOR_DO` as *`scan`*.

END_FOR_DO

Syntax

```
END_FOR_DO
```

Operation

Completes a `BEGIN_FOR_DO` declaration.

Example

This example extends the Rational DOORS C API for a new language, TXL. It declares a function `tdsCreate`, which appears as create in a TXL script. It takes a TXL string parameter (a `char*` in C) and returns a TXL Table value (a `Table*` in C).

```
apiInstall("void create(string)",tdsCreate)
BEGIN_FN(tdsCreateFn,1,1)
    P_(char*,name);
    Table* tab;
    END_DECLS;
    tab = tdsCreate(name);
    RETURN_(tab);
END_FN
```

This example creates a `for..do` loop. Entry is the TXL data type representing a C `Entry*` variable, and is the type of the `scan` variable. `Table` is the TXL data type representing a C `Table*` variable, and is the parent of the scan.

```
BEGIN_FOR_DO(tdsDoFn,Table*,tab,Entry*,scan)
    tdsDo(tab,scan) {
      PROCESS_DO(scan);
    }
END_FOR_DO
apiInstall("void ::do(Entry&, Table, void)",
             tdsDoFn);
```

Given these declarations you can run the TXL script:

```
Table tab = create "my table"
tab["1"] = "one"
Entry e
for e in tab do {
    print (key e) "\n"
}
```

The `PROCESS_DO` macro causes the code:

```
print (key e) "\n"
```

to be executed for each `Entry e`. The code:

```
tdsDo(tab,scan)
```

of `tdsDoFn`, causes `scan` to be set to each `Entry*` in `tab`, which in turn appears as `e` in the TXL script.

Rational DOORS C API entry points

In the entry points that follow, the parameters of external function declarations are shown within `#if` and `#endif` statements.

apiError

```
extern void apiError();
#if 0
    char *format;
    ...
#endif
```

Causes the calling program to exit and issue an error message. The parameter *format* is a printf style format. If only one parameter is used, the character % must appear as %%.

apiWarn

```
extern void apiWarn();
#if 0
    char *format;
    ...
#endif
```

Issues a warning message. The parameter *format* is a printf style format. If only one parameter is used, the character % must appear as %%.

apiMainProg

```
extern void apiMainProg();
#if 0
   int argc;
   char* argv[];
   char* name;
   char* ext;
   char* include;
   void (*init)();
   void (*done)();
#endif
```

Sets up a Rational DOORS active link main program.

The arguments *argc* and *argv* are the normal C main program parameters.

The argument *name* is the name of the resulting language (for example, TXL). A null value causes the default core DXL Interpreter (CDI) to be used.

The argument *ext* is the file extension used by scripts (for example, .txl). A null value causes the default, .cdi, to be used.

The argument *include* is a separate path of places to search for source and include files. A null value defaults to the current directory.

The function init should contain all the initialization needed for the server.

The function done should do all the final winding down for the server.

apiInitLibrary

```
extern void apiInitLibrary();
#if 0
   char* n;
   char* ext;
```

```
   char* include;
#endif
```

Initializes the API when `apiMainProg` is not being used. The parameters are as described in apiMainProg.

apiFinishLibrary

```
extern void apiFinishLibrary();
#if 0
#endif
```

Winds down the API.

apiParse

```
extern void apiParse();
#if 0
    char *format;
    ...
#endif
```

Parses and executes the parameters in the API's interpreter. The parameter *`format`* is a `printf` style format. If only one parameter is used, the character `%` must appear as `%%`.

For examples of the use of `apiParse`, see "Listing of tds.c," on page 49.

apiConnectSock

```
extern void apiConnectSock();
#if 0
    unsigned short portNum;
    char* hostAddr;
#endif
```

apiSend

```
extern void apiSend();
#if 0
    char *format;
    ...
#endif
```

Sends the specified string down the connection made with `apiParse` or `apiConnectSock` as a DXL script to be executed by Rational DOORS. The parameter *`format`* is a `printf` style format. If only one parameter is used, the character `%` must appear as `%%`.

A subsequent call to `replyAPI`, causes `apiSend` to execute the string passed to `replyAPI` using the API's interpreter.

apiSendTimesout

```
extern void apiSendTimesout();
#if 0
    int tmt;
    char *format;
    ...
#endif
```

Like `apiSend`, but the *tmt* parameter is the number of seconds it waits for the reply. The parameter *format* is a *printf* style format. If only one parameter is used, the character `%` must appear as `%%`.

apiSendFile

```
extern void apiSendFile();
#if 0
   char *f;
#endif
```

A file variant of `apiSend`, which sends the file pointed to by *f* as a DXL script to be executed by Rational DOORS.

apiExitOnError

```
extern void apiExitOnError()
#if 0
   int onOff;
#endif
```

Sets whether the API functions exit whenever there is an error. By default, the functions exit, but you can prevent that using this function.

apiQuietError

```
extern void apiQuietError()
#if 0
   int onOff;
#endif
```

Sets whether the API functions produce error messages on the command line. By default, the functions produce command line error messages, but you can prevent that using this function.

apiGetErrorState

```
extern int apiGetErrorState()
#if 0
#endif
```

Returns the error that occurred most recently. Possible return values are:

```
DOORS_API_OK
DOORS_API_PARSE_BAD_DXL
DOORS_API_SEND_BAD_DXL
DOORS_API_CONNECT_FAILED
DOORS_API_ERROR
```

apiGetIPC

```
extern void *apiGetIPC()
#if 0
#endif
```

Returns a pointer to the IPC channel currently being used by the API.

apiSetIPC

```
extern int apiSetIPC()
#if 0
   void *newIPC;
#endif
```

Sets the IPC channel for use by the API. Returns 1 if `newIPC` was set; otherwise, returns 0. Returns 0 if `newIPC` is null or not connected.

apiDeleteIPC

```
extern void apiDeleteIPC()
#if 0
   void *IPC;
#endif
```

Deletes the specified IPC channel.

4 Using the DXL server

This chapter describes how to use the DXL server, which allows external applications to send DXL programs to Rational DOORS for execution. It contains the following topics:

- About the DXL server interface
- Using the DXL server in batch mode

About the DXL server interface

The DXL server allows programs external to Rational DOORS to send DXL messages to Rational DOORS for execution. For example, a Windows Command prompt could send messages to Rational DOORS.

The DXL server interface consists of two programs:

```
dxlips
dxlipf
```

The `dxlips` and `dxlipf` programs use TCP/IP port and host sockets to connect to Rational DOORS.

Note The DXL server can only be launched from an interactive Rational DOORS session. It is not supported from batch DXL programs. The behavior of the DXL server can be emulated from batch DXL using the DXL program described later in this chapter.

If an external tool allows commands to be invoked from within its user interface, these programs can be used to communicate with Rational DOORS. An example of such an external tool is a CASE tool that has a user-defined menu.

Rewrite these programs for your own tool.

Starting the server

On all platforms, executing the following DXL from the DXL Interaction window starts the TCP/IP server on port 5093 (the default port):

```
evalTop_ "initDXLServer server 5093"
```

Alternatively, the line:

```
initDXLServer server 5093
```

could be included in `startup.dxl`.

Note In practice, do not hard code port numbers. Instead make sure that they can be configured by the user.

After initializing the server, you can use the server interface commands. This level of tool integration does not directly support receiving replies from Rational DOORS.

The commands `dxlips` and `dxlipf` are simple utilities that use the Rational DOORS C API facilities described in “The Rational DOORS C API,” on page 7. The source code is supplied in `$DOORSHOME/api`.

dxlips

The `dxlips` program is supplied with Rational DOORS in `$DOORSHOME/bin`.

It takes a single string command-line argument, which is sent to Rational DOORS and interpreted as a DXL program.

Rational DOORS and `dxlips` can be run on different machines. They communicate through a TCP/IP socket with a default port number given by the environment variable `DXLPORTNO` on a host indicated by `DXLIPHOST`. The server always runs on the same host as Rational DOORS.

Example

This example of `dxlips` causes the date on which the current Rational DOORS session started to be printed in the Rational DOORS DXL Interaction window’s output pane.

```
%DOORSHOME%\bin\dxlips "print session"
```

dxlipf

The `dxlipf` program operates in the same way as `dxlips`, except that the command-line argument specifies the name of a file which contains a DXL program to be sent to Rational DOORS.

system

Rational DOORS allows external tools to be called using the DXL command system, which is described fully in the *DXL Reference Manual*.

Usage

```
system("C:\winnt\system32\command /c dir")
```

You can call the system command several times in the same script. Each time it is called a new process is forked to run the command.

If you run more than twenty processes, the behavior is undefined. To avoid this, ensure that each group of fewer than twenty system commands has adequate time to complete before you move on to the next group.

One way to do this is to place an `ack` command between each group of calls.

Using the DXL server in batch mode

Rational DOORS has two modes of operation: **interactive mode**, where there is a graphical user interface, and **batch mode** where Rational DOORS runs with no graphical user interface.

To run Rational DOORS in batch mode, at the prompt type:

```
doors -batch dxlfile
```

The built in DXL server started by the `initDXLServer` function cannot be used in batch mode. As an alternative, to emulate the built in server, you can use the following script modified to meet the requirements of the interface being written:

```
// batchserver.dxl
IPC ipc = server 5093
string request
/* add functions for your interface here */
while (true) {
  if (accept(ipc)){
    if (!recv(ipc,request)) {
      warn "Server has disconnected"
      break
    }
  }else{
    warn "error accepting client connection"
    break
  }
  print "request: "
  print request
  print "\n"
  errors=false
  if (request=="shutdown_"){
    send(ipc,"done_")
    break
  }
  if (request=="errors_")
    break
  if (request=="quit_")
    continue
  ans = eval_ request
```

```
  if (ans=="errors in eval_ string") {
    print "errors in request\n"
  }
  send(ipc,"done_")
  disconnect(ipc)
}
```

5 DXL API integration features

This chapter describes DXL features required by the integration engineer. They are omitted from the *DXL Reference Manual* because they are potentially hazardous.

This chapter contains the following topics:

- General functions
- Interprocess communications
- DXL contexts

General functions

*addr_*

Syntax

```
addr_(y)
```

Operation

Takes arguments of any type and returns them in any context, for example:

```
bool x  = addr_ 1
bool y  = addr_ 0
print x " " y "\n"
```

Prints `true false`.

Note This function is extremely hazardous, as it allows the type system of DXL to be violated. Use it with care, if you *must* override DXL types.

*eval_*

Syntax

```
string eval_(string)
```

Operation

This function causes its parameter to be executed by the DXL interpreter, within a private context. Declarations made within the execution do not persist after the execution is complete. The result is a string which can be set using the `return_` function.

*return_*

Syntax

```
void return_(string)
```

Operation

When used within a string passed to `eval_`, makes its argument the result of the call to `eval_`.

*evalTop_*

Syntax

```
string evalTop_(string)
```

Operation

Like `eval_`, but executes within the outermost context of the DXL interpreter, thus making any declarations persist. When an `evalTop_` call appears in a DXL script its argument is not executed until the enclosing script has finished executing.

The following script produces an error:

```
evalTop_("int a_ = 3")
print a_
```

When you place a variable or function in the top context, take care to avoid clashes with variables in other DXL programs. The name of such a variable should have a prefix that is the name of the tool in which it is used, and a suffix of an underscore. For example, for TDS you could use `TDS_IPC_`.

initDXLServer

Syntax

```
void initDXLServer(IPC dxlsrvr)
```

Operation

Initializes the DXL server, using a TCP/IP socket to communicate. The IPC channel can be initialized by the `server` function.

replyAPI

Syntax

```
void replyAPI(string reply)
```

Operation

Sends the passed string back to the DXL server. This is useful in code that is called by DXL server clients using the `apiSend` function.

setAPIClientTimeout

Syntax

```
void setAPIClientTimeout(int tmt)
```

Operation

Sets the time limit for the `replyAPI` function to wait for an acknowledgement from the DXL server.

ipcHostname

Syntax

```
string ipcHostname(string hostAddr)
```

Operation

Returns the name of the host with IP address *`hostAddr`*.

ipcAddress

Syntax

```
string ipcAddress(string hostName)
```

Operation

Returns the IP address of the host named *`hostName`*.

Interprocess communications

The following functions provide interprocess communication operations:

server

Syntax

```
IPC server(int portno)
```

Operation

Establishes a server connection to port number *portno*.

client

Syntax

```
IPC client(int portno,
           string host)
```

Operation

Establishes a client connection to IP address *portno* at *host*.

accept

Syntax

```
bool accept(IPC chan)
```

Operation

Waits for a client connection. This is used by servers.

send

Syntax

```
bool send(IPC chan,
          string message)
```

Operation

Sends the string *message* down the IPC channel *chan*.

recv

Syntax

```
bool recv(IPC chan,
       {string|Buffer} &response
        [,int tmt])
```

Operation

Waits for a message to arrive in channel *chan* and assigns it to string or buffer variable *response*.

The optional third argument defines a time-out, *tmt* seconds, for a message to arrive in channel *chan*. If *tmt* is zero, this function waits forever. It only works if the caller is connected to the channel as a client or a server.

DXL contexts

To avoid over-use of resources, every function and variable declared in DXL has a finite lifetime. When it is no longer being used the memory that it was allocated is freed. The lifetime of a variable depends on the lifetime of the context in which it is declared.

If you attempt to access variables and functions outside their lifetimes, the results are undefined, but may cause Rational DOORS to fail. There are two types of context:

- Top context

  Code included in `startup.dxl` or executed by the `evalTop_` function is in the top context.

- Local context

  Code run from a menu, the DXL Interaction window or a call to the `eval_` function, runs in its own local context.

Programs run in local contexts can access names declared in the top context. A local context is deleted when all dialog boxes created by the program run from the context are closed down. A program that is run in a local context and does not create any dialog boxes has its resources reclaimed after it terminates.

A common mistake is shown in the following scripts.

First script:

```
evalTop_("DB db_");
```

Second script:

```
void callback(DBE b){
  ack "button pressed"
}
db_ = create "Test DB"
DBE b = button(db_, "Fail", callback)
```

Third script:

```
show db_
```

By the time the third script is run, the memory occupied by the dialog box db and its callback function has been freed and the behavior is undefined. To make these scripts work, the second script must run in the top context.

Impact on triggers

Dynamic triggers are governed by the same context rules as variables and functions. When you set a dynamic trigger in a DXL script, it is deleted when the script finishes, and you do not see its effect. There are two ways to make the dynamic trigger survive:

- Place it in the top context using the evalTop_ function, taking care to avoid name clashes.
- When the trigger is related to a DXL dialog box, keep the dialog box open.

Consider the following script:

```
bool dynTrig(Trigger t){
    ack "closing"
    return true
}
trigger(module, close, 10, dynTrig)
DB db = create "test"
show db
```

With a formal module open, run the script. Close the formal module and the trigger fires. Close the test dialog box and then re-open the formal module. Run the script again. Close the test dialog box and then close the formal module. The trigger does not fire.

In the first case the trigger fired because the context of the DXL script it was declared in was still open. In the second case the context had been closed when the dialog box was closed, so the trigger was no longer present and so did not fire.

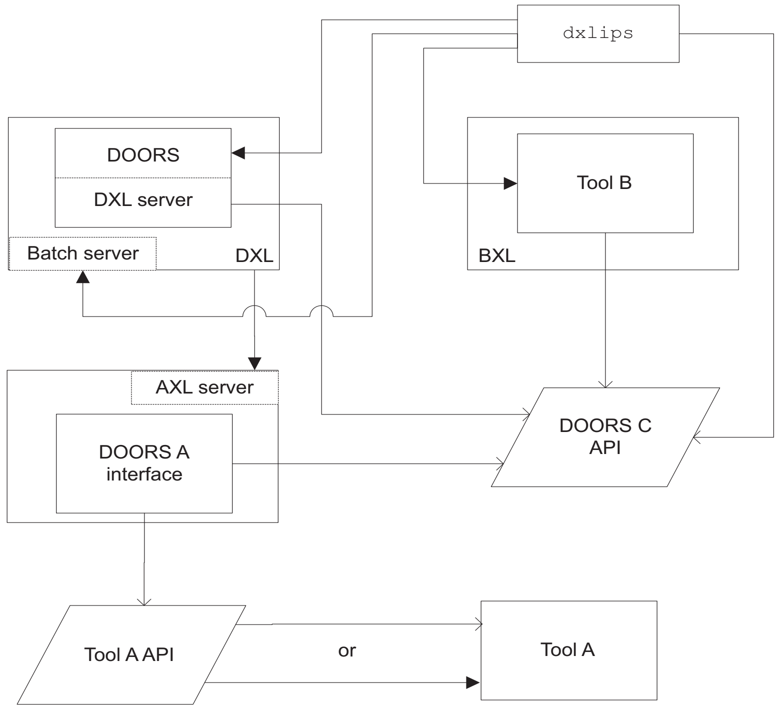
6 Interactive interfacing with a complex external tool

This chapter describes how to build interfaces between Rational DOORS and other tools, such as CASE tools or other complex packages. It contains the following topics:

- Integrating Rational DOORS with user tools
- Integrating Rational DOORS using Rational DOORS URLs
- Example tool to be interfaced to Rational DOORS
- Working with OLE objects
- Listing of tds.c

Integrating Rational DOORS with user tools

The powerful requirements analysis, manipulation and presentation facilities provided by Rational DOORS can be exploited to an even greater extent if Rational DOORS is tightly coupled to the other tools present in the user's own environment. Rational DOORS uses its extension language, DXL, to provide the basis for such links, together with the Rational DOORS C API which enables users to build DXL-like languages around existing tools and also provides the interprocess communication facilities needed to establish a link with Rational DOORS. Using the extension language to build an interface layer around both Rational DOORS and user tools is a powerful and flexible tool linkage strategy. The strategy is shown in the following diagram.



Programs are represented by boxes and code libraries by parallelograms. Light headed arrows between boxes are C function calls. Heavy headed arrows are IPC communications. Both types of call can be used to read or write data in both directions. The arrowhead direction indicates who initiates the call.

The left half of the diagram represents a connection to an external tool `A` that provides an API (a set of functions that can be called in to input or output data to or from the tool. The program `DOORS A interface` interfaces with tool `A`'s API and communicates with Rational DOORS via an IPC channel. In this configuration the external tool is acting as a server and Rational DOORS as its client (a Rational DOORS active link).

The right half of the diagram represents tool connections where Rational DOORS is expected to serve calls from the external tool (a Rational DOORS passive link). In this configuration, Rational DOORS acts as the server (using the DXL server) and the external tool acts as the client.

Both types of link make use of the Rational DOORS C API, as do Rational DOORS and the server utilities. Most of the code required to establish a link between tools is written using either DXL or, in the case of an active link, a

DXL-like language created for the external tool's API (AXL). The Rational DOORS C API supports the construction of this language and its interpreter.

The overall strategy for a Rational DOORS active link is:

1. Using the Rational DOORS C API, create a DXL-like language to interface to the target tool's API. These bindings form the major part of the Rational DOORS active link.
2. Create DXL scripts for execution by Rational DOORS that implement the command set of the desired link. This typically involves writing DXL functions that send data to the external tool, and writing DXL functions that can be called by the external tool to send results back to Rational DOORS.
3. Create scripts for execution in the Rational DOORS active link (AXL scripts in this example) that implement the command set of the desired link.
4. Run the Rational DOORS active link as a server process. Commands made available by stages 2 and 3 can now be executed from Rational DOORS, typically through DXL generated menus and forms.

The overall strategy for a Rational DOORS passive link is:

1. Determine what messages need to be sent to and from Rational DOORS. Render these messages as DXL function calls.
2. Create DXL scripts that implement the bodies of the function call messages of Step 1. Functions to be executed on the external tool (client) side need to be installed as DXL extensions using the Rational DOORS C API.
3. Link (in the C object library sense) the Rational DOORS API to the external tool.
4. Start a Rational DOORS DXL server to handle requests from the external tool.

To illustrate the tool linkage strategy the following section uses an example target application: the Toy Database Server (TDS). The example shows the development of both a Rational DOORS active and a Rational DOORS passive link.

Integrating Rational DOORS using Rational DOORS URLs

This section is for integrators who want to refer to Rational DOORS resources.

A Rational DOORS URL has the following syntax:

```
doors://<hostport>/?<search_specification>
```

- Where <*hostport*> is the host name and port number of the Rational DOORS database server that contains the Rational DOORS resource. For example, `server.domain:36677`.

  Note You must provide the port number.

- <*search_specification*> defines the resource. It is a comma-separated list of search elements. The search elements and their meanings are as follows:

Search element	Meaning
`dbid=<unreserved>`	The identifier of a database. This is mandatory in version 1 of the URL.
`version=<version>`	<*version*> is an `<unreserved>` that represents the version of the URL syntax. The version numbering scheme is non-zero natural numbers from 1. In Rational DOORS 9.2, the version number of all Rational DOORS URLs will be 1. That is: `version=1`
prodID=<*nat*>	<*nat*> is a `<reserved>` that is the decimal representation of a natural number. This is used to indicate the product that generated the URL. the current permitted values is: • 0 - Rational DOORS
`container=<unreserved>[":"<version>]`	`The identifier of a container (for example, project, folder or module) within the database. Notice that this identifier may include version information where <version> ::= <unreserved>`
`object=<unreserved>`	The identifier of an object (within a document container). In case of Rational DOORS 9.2 URLs, the object is denoted by its **Absolute Number** attribute. For example `object=23`.

Examples of Rational DOORS URLs

- A database URL. Opening a database URL causes the root of the database to be displayed in the database explorer.

  ```
  doors://greenback:36677/?version=2&prodID=0&urn=urn:telelogic::
  1-49d22a0e60b71ecc-A
  ```

- A project URL. Opening a project URL causes the project to be made current in the database explorer.

  ```
  doors://greenback:36677/?version=2&prodID=0&urn=urn:telelogic::
  1-49d22a0e60b71ecc-P-00000020
  ```

  The `-P-` in the URL denotes a project.

- A folder URL. Opening a folder URL causes the folder to be made current in the database explorer.

  ```
  doors://greenback:36677/?version=2&prodID=0&urn=urn:telelogic::
  1-49d22a0e60b71ecc-F-00000046
  ```

  The `-F-` in the URL denotes a folder.

- A module URL. Opening a module URL causes the module to be opened in the default edit mode with the default view displayed.

  ```
  doors://greenback:36677/?version=2&prodID=0&urn=urn:telelogic::
  1-49d22a0e60b71ecc-M-000000a0
  ```

  The `-M-` in the URL denotes a module.

- An object URL. Opening an object URL causes the containing module to be opened in the default edit mode with the default view displayed and the specified object selected. The normal view changing rules apply if the object is not displayed in the view.

  ```
  doors://greenback:36677/?version=2&prodID=0&urn=urn:telelogic::
  1-49d22a0e60b71ecc-O-4-000000a0
  ```

  The `-O-` in the URL denotes an object.

Example tool to be interfaced to Rational DOORS

TDS is a very simple table manipulation package. Tables can be created and deleted, and their entries created and deleted. Although a small program, it exercises all the major features of a more complex Rational DOORS link program.

C API for example

The C API for TDS is for use with a Rational DOORS active link. It is the set of C data structures and entry points that it provides to be called by interfacing programs.

```
/*
*   Data Structures:
*/
typedef struct Table_ Table;
typedef struct Entry_ Entry;
struct Table_ {
   string name;
   Entry* es;
   Table* next;
   int size;
};
struct Entry_ {
   string key;
   string data;
   Entry* next;
};
```

A table of type `Table` is simply a linked list of entries of type `Entry`. All tables are linked together.

APIs often have exit codes defined as function results, as in this example:

```
#define StatusOK 0
#define StatusBadDeleteEntry 1
#define StatusBadDeleteTable 2
```

The two macros below define traversal macros for the two data structures:

```
#define tdsDo(table,e) for (e=table->es; e != NULL; e = e->next)
#define tdsTabDo(t) for (t=AllTables; t != NULL; t = t->next)
```

All tables are linked and accessible from this variable:

```
externvar Table* AllTables;
```

To describe the remaining functions of the C API for TDS, the parameters of external function declarations are shown within `#if` and `#endif` statements.

Entry Point	Use

`extern Table* tdsCreate();`
`#if 0`
`   string s;`
`#endif`
	Creates a table with name `s`.

Entry Point	Use

`extern Entry* tdsEntry();`
`#if 0`
`   Table* t;`
`   string key;`
`   bool create;`
`#endif`
	Looks up the entry according to key in table `t`. If the entry does not exist and create is true then create it.

`extern void tdsPut();`
`#if 0`
`   Entry* e;`
`   string data;`
`#endif`
	Associate the string data with the entry `e`.

`extern string tdsGet();`
`#if 0`
`   Table* t;`
`   string key;`
`#endif`
	Returns the data for the given table `t` and key. If key does not exist, returns a null string.

`extern int tdsDeleteEntry();`
`#if 0`
`   Table* t;`
`   string key;`
`#endif`
	Deletes the entry specified by `t` and key.

`extern int tdsDeleteTable();`
`#if 0`
`   Table* t;`
`#endif`
	Deletes the given table.

`extern void tdsInfo();`	A diagnostic routine.
`extern void tdsInit();`	An initialization routine.
`extern void tdsFinish();`	A final housekeeping entry point.

This completes the API for TDS. The implementation of this interface is in `$DOORSHOME/api/tdsfns.c`.

Making a language like DXL

Using the interface presented in "C API for example," on page 32, you can now make a language like DXL to drive the interface: a Rational DOORS active link. From the basis of the core DXL language, you can add TDS specific data types and commands. For this exercise, the resulting language is called TXL, and the extension `.txl` is used on files containing TXL scripts.

The program `$DOORSHOME/api/tds.c` fully implements a DXL-like interface to TDS. All the Rational DOORS C API entry points are described in "The Rational DOORS C API," on page 7. The complete source for `tds.c` is given in "Listing of tds.c," on page 49. Extracts from this program illustrate how to build the language.

Including files

After some comments, the program begins with the following include statements:

```
#include <doors/api.h>  /* API services */
#include "tds.h"  /* this file's entry points */
#include "tdsfns.h"     /* the TDS API */
```

The first include statement is the normal way of accessing the Rational DOORS C API from within a C program. The makefile given for TDS (also in `$DOORSHOME/api`) shows one way of specifying where to find both the include file and the necessary API object file.

Declaring functions

After including the necessary `.h` files, `tds.c` continues with:

```
BEGIN_FN(tdsCreateFn,1,1)
   P_(char*,name);
   Table* tab;
   END_DECLS;
   tab = tdsCreate(name);
   RETURN_(tab); /* return the created table */
END_FN
```

The macro `BEGIN_FN` takes three parameters: the name of the C function to be registered with the API, the number of input parameters and the number of results (either 0, corresponding to void, or 1).

The line `P_(char*,name)` specifies that the first parameter is of type `char*` and is called `name`. After specifying all parameters (there are no more in this example), you must also declare any variables to be used in the function being defined. `END_DECLS` marks the end of declarations, and is always needed. The body of the function calls `tdsCreate` with the passed name and returns the result. The macro RETURN_ indicates what the DXL-like function should return when executed, but does not return from the function. END_FN ends the declaration of the new DXL-like function.

Installing functions

Later in tds.c there are the following lines:

```
apiParse("struct Table {}; struct Entry {};");
apiInstall ("Table create (string)",
              tdsCreateFn);
```

This is the second part of registering a new function for a DXL-like language. The first parameter of apiInstall is the prototype of the new function, which must match the information supplied for numbers of parameters and results given to BEGIN_FN. The second parameter is the name of the function created using BEGIN_FN. In the DXL-like language you are building, the function is called create.

The function apiParse parses and runs its parameter. In this case it is the definition of two new data types for TXL: Table and Entry. Refer to “DXL API integration features,” on page 21 for more information.

The program tds.c continues by specifying many more DXL-like commands in this way. Effectively, it makes a link from a C function (here tdsCreate), to the Rational DOORS C API’s interpreter, using an intermediate function (here tdsCreateFn).

Declaring and installing a for loop

Later in tds.c there is:

```
BEGIN_FOR_DO(tdsDoFn,Table*,tab,Entry*,scan)
     tdsDo(tab,scan) {
        PROCESS_DO(scan);
    }
END_FOR_DO
```

This fragment should be considered paired with the later:

```
apiInstall("void ::do (Entry&, Table, void)",
                         tdsDoFn);
```

The macro BEGIN_FOR_DO allows you to provide a DXL-like for loop for TXL. Its parameters are:

- The name of the function: tdsDoFn
- The type of the parent of the loop: Table*
- A variable in which the parent is to be placed; the parent is some variable from which you can initialize the loop

- The type of the elements of the loop: `Entry*`
- A variable in which each element in turn is to be placed

The `tdsDo` macro is defined in "C API for example," on page 32.

The macro `PROCESS_DO` makes the currently scanned element available to the body of the loop.

The call to `apiInstall` defines a function that returns void and has three parameters. The first parameter is a reference type for the scanned element; the second parameter is the parent; the third parameter is void. The installation of a for loop must always be in this format.

Main program

The final step in making a DXL-like language is the main program:

```
/* main.c
 * The main program of the DXL-like
 * language, TXL  */
#include <doors/api.h>
#include <stdio.h>
#include "tds.h"
extern char* getenv();
int main (argc, argv)
    int argc;
    char* argv[];
{
    static char path[255];
    sprintf(path,"%s/lib/txl",
                          getenv("DOORSHOME"));
    apiMainProg(argc, argv, "TXL", ".txl", path,
                          tdsInitAPI, tdsFinishAPI);
    return 0;
}
/* end of main.c */
```

`apiMainProg` has the following parameters:

- The normal C main argument, `argc`
- The normal C main argument, `argv`
- The name of the language being built, `TXL`, as a string
- A default file extension, `.txl`, as a string
- A default search path for source and include files, `path`
- An initialization function (called by `apiMainProg`)
- A termination function (called by `apiMainProg`)

The file `tds.c` implements a small, but powerful, DXL-like language for TDS. The command line arguments for the language are the file names of scripts containing TXL programs.

Building the object file

To build the object file, see "Compiling TXL with Microsoft Developer Studio," on page 38.

Executing a TXL script

After the object file has been created, the following TXL program can be executed:

Example TXL script:

```
void printTab (Table t) {
    Entry e
    print "(" (name t) ":\n"
    for e in t do        // the tdsDoFn loop
        print (key e) " : " (data e) "\n"
    print ")\n"
}
void printAll () {
    Table t
    for t in All do
        printTab t
}
void doDelete (Table t, string key) {
    int status
    delete (t, key)
    if (status !=StatusOK)
        warn "no record for " key " in " (name t)
}//doDelete
Table t = create "english2french"
// the tdsCreateFn  function
t["one"]      =  "un"
t["two"]      =  "deux"
t["three"]    =  "trois"
t["four"]     =  "quatre"
t["five"]     =  "cinq"
t["six"]      =  "six"
t["seven"]    =  "sept"
t["eight"]    =  "huit"
t["nine"]     =  "neuf"
t["ten"]      =  "dix"
print t["three"] "\n"
print "----------\n"
printTab t
```

```
doDelete (t, "two")
print "-----------\n"
printTab t
info
```

A similar script is in `$DOORSHOME/lib/txl/tds.txl`.

Compiling TXL with Microsoft Developer Studio

This section describes how to build `txl.exe` using Microsoft® Developer Studio. The executable file can be built using any C compiler and you should adapt these instructions for your own environment.

1. Select **File > New > Project workspace**, and then choose **Console Application** with the name `txl` in directory `%DOORSHOME%/api`. (`%DOORSHOME%` is the directory pointed to by `HOME` in your `doors.ini` file.)
2. Select **Insert > Files into project**. Add `tdsfns.c`, `tds.c`, and `main.c`.
3. Select, **Build > Setting > Link**. In the dialog box add `dxlapi.lib` to the Object library modules field, and `dxlapi.dll` to the path.
4. Select **Tools > Options**, then select the **Directories** tab. Add `%DOORSHOME%/include` to the include files directories and `%DOORSHOME%/bin` to the library files directories.
5. Press **F7** to build `txl.exe`.

Completing the Rational DOORS active link

Now that you have a DXL-like language for TDS, you can use it to build a command server for Rational DOORS.

DXL includes IPC facilities that allow messages to be passed between DXL interpreters. Rational DOORS can send messages to the TXL interpreter to be executed, and vice versa. This is a simple and effective example of a client/server architecture.

To complete the Rational DOORS active TDS link:

1. Using the DXL library (click **Tools > Edit DXL > Browse**), locate the Rational DOORS client for TDS. The source is in `$DOORSHOME/lib/dxl/example`:

   The code in `apiinit.dxl` initializes the TDS server; the code in `apistart.dxl` starts the TDS interaction window.
2. Run `apiinit` and then `apistart`. The Rational DOORS/TDS Link window is displayed.

It has the following buttons:

Button	Function
start server	Starts the TDS server as a process in an xterm or DOS shell.
add current heading	Sends the current Rational DOORS object heading, and the name of the user who created the object, to TDS for inclusion in a TDS table as key and data.
delete current heading	Sends the current Rational DOORS object heading as a key to delete an entry in a TDS table.
print table	Prints the TDS table, and sends each entry to Rational DOORS for display in a popup window.
shutdown server	Shuts down the TDS server, causing the xterm or DOS shell to exit.
close	Closes the window.

The start server button executes `$DOORSHOME/api/txl.exe`, with the same arguments as above.

The server uses a simple protocol. It opens up an IPC server on the named socket, and waits for connections from Rational DOORS clients. Rational DOORS makes a connection via the `start server` command, which is issued by the **start server** button. The messages sent by Rational DOORS are implemented in the included file `t2d.txl`, which is in the same directory as `server.txl`.

Code in server.txl

```
// TDS server
IPC ipc = server port       // port is passed in by api.inc
bool debug=false            // true => diagnostic output
bool errors=false           // have we had an error?
void dprint(string s) {     // diagnostic routine
    if (debug) print s
}
void toDoors (string s) {
// send message, must be acknowledged
    dprint "toDoors(" s ")\n"
    if (!send(ipc, s))
        unixerror "toDoors/send"
```

```
    if (!recv(ipc, s))
        unixerror "toDoors/recv"
    dprint "Ack: " s "\n"
    if (s!="OK") eval_ s
}//toDoors
void done () {
// send "done" message, no acknowledge needed
    dprint "done\n"
    if (errors)
        // client has already disconnected
        errors=false
    else {
        if (!send(ipc, "done_"))
            unixerror "done/send"
    }
}//done
void sendError (string mess) {
// let Rational DOORS know about an error
    if (errors)
        return
    if (!send(ipc, "errors_"))
        unixerror "error/send"
    if (!recv(ipc, s)) // the ack
        unixerror "toDoors/recv"
    if (!send(ipc, mess))
        unixerror "error/send"
}//sendError
string request
string res
#include <t2d>
checkIPC ipc
// must be provided in client specific part
print "Ready to accept commands from DOORS\n"
if (!accept ipc)
    unixerror "unexpected failure waiting for
                DOORS client"
while (true) {
    if (!recv(ipc,request)) {
        warn "DOORS has disconnected"
        break
    }
    dprint "request: " request "\n"
    errors=false
    if (request=="shutdown_")
        break  // no acknowledge needed
    ans = eval_ request
    if (ans=="errors in eval_ string") {
        print "errors in request\n"
```

```
        done
    }
    if (request=="shutdown_")
        break  // no acknowledge needed
}//while (true)
closeDown
// must be provided in client specific part
// tds/doors interface
#include <utils>
/*
  data
*/
Table doors = create "doorsTable"
/*
  the following commands are sent by doors for execution by tds
*/
void associate(string s1, s2) {
    print "receiving key \"" s1 "\" with data \""
            s2 "\"\n"
    doors[s1] = s2
    done
}
void delete(string s1) {
    int status = delete(doors,s1)
    print "deleting \"" s1 "\"\n"
    if (status ! = StatusOK)
        sendError "Heading \"" s1 "\" not in table"
    else
        done
        // don't do "done" if we have an error
}
void list() {
    Entry e
    int i=0
    printTab doors
    for e in doors do {
        i++
        toDoors "fromTds(\"" (key e) "\", \""
                  (data e) "\")"
    }//for
    if (i==0) {
        sendError "no entries"
        return     // no done needed
    }
    done
}
/* server needs these two entrypoints */
void checkIPC(IPC ipc) {
    if (null ipc)
```

```
        unixerror "unexpected failure creating
                     TDS server "socket"
}//checkIPC
void closeDown() {
    print "server shutdown \ n"
}
```

Initializing the client

The Rational DOORS client side of the link is initialized by `apiinit.dxl`, which is in `$DOORSHOME/dxl/example/apiinit.dxl`. It contains the following statement:

```
evalTop_ "#include <example/api.inc>"
```

The internal DXL `evalTop_` makes any definitions available to further executions of DXL programs.

Except for `startup.dxl`, a DXL program runs in its own private context. Refer to “DXL API integration features,” on page 21 for an explanation of DXL’s context rules.

The file `api.inc` contains the following:

```
/*
  Rational DOORS API Demo
  $DOORSHOME/lib/dxl/example/api.inc
*/
#include <utils/unique>
IPC tdsIPC = null
string tdsName = "DOORS/TDS"
int port = 5097
string host = "127.0.0.1"
string dhome = getenv "DOORSHOME"
bool tdsDebug=false
void tdsDprint(string s) {
    if (tdsDebug) cout << s
}
void tdsError (string mess) {
    tdsDprint mess
    ack tdsName ": " mess
    halt
}
void ackRecv() {
    if (!send(tdsIPC, "ack"))
        tdsError "ackRecv_ failed"
}
void tdsSend (string request) {
    string response, res
    tdsDprint ">> " request "\n"
```

```
    if (!send(tdsIPC, request))
       // tdsSend client request
        tdsError "tdsSend failed"
    if (request=="quit_" || request=="shutdown_")
        return
    while (true) {
        if (!recv(tdsIPC, response))
            tdsError "recv failed"
        tdsDprint "< " response "\n"
        if (response=="done_")
            // computation completed
            break
        if (response=="errors_") {
            // error message
            ackRecv
        if (!recv(tdsIPC, response))
            tdsError "recv failed"
            tdsError "tds server failure:  "
                       response
        }
        res = eval_ response
        if (res=="")
            res = "OK"
        tdsDprint "> " res "\n"
        if (!send(tdsIPC, res))
             // need response until "done" sent
             tdsError "tdsSend failed"
    }
}
bool connected () {
    if (null tdsIPC) {
        tdsIPC = client(port,host)
        if (null tdsIPC) {
            ack "not connected yet"
            return false
        }
    }
    return true
}
/* Dialogue box stuff */
DB TDS=null
DBE tdsB1, tdsB2, tdsB3, tdsB4, tdsB5
bool TDSIsShowing = false
void finishTDS(int status) {
    tdsIPC = null
}
void tdsF1(DBE dbe) {
    if (!(null tdsIPC)) {
        ack "server socket already exists"
```

```
    } else{
        if (platform == "WIN32") {
            if (!fileExists_(dhome "/api/txl.exe")){
                ack("You must first make txl.exe
                     in $DOORSHOME/api. \nSee
                     the DOORS API Manual for
                     details.")
                return
            }//!fileExists_
            system(dhome "/api/txl  " dhome
                   "/lib/txl/server.txl int port = " port " ")
        }else{
            if (!fileExists_(dhome "/api/txl")){
                ack("You must first make txl in
                     $DOORSHOME/api. \nSee the
                     DOORS API Manual for
                     details.")
                return
            }//!fileExists_
            system("xterm -e " dhome " /api/txl  "
                   dhome "/lib/txl/server.txl  int
                  port = " port  " ", finishTDS)
        }
    }//else
}//tdsF1
void tdsF2(DBE dbe) {
    string h =
           (current Object)."Object Heading" ""
    string b = (current Object)."Created By" ""
    if (connected)
        tdsSend "associate(\"" h "\", \"" b "\")"
}
void tdsF3(DBE dbe) {
    string h = (current Object)."Object Heading"
                 ""
    if (connected)
        tdsSend "delete \"" h "\""
}
void tdsF4(DBE dbe) {
    if (connected)
        tdsSend "list"
}
void tdsF5(DBE dbe) {
    if (connected) {
        tdsSend "shutdown_"
        tdsIPC = null
    }
}
```

```
void closeCB(DB db){
    TDSIsShowing = false
    hide db
}
void initTDS () {
    TDS = create "DOORS/TDS Link Control"
    tdsB1 = button(TDS, "start server", tdsF1)
    tdsB2 = button(TDS, "add current heading",
            tdsF2)
    tdsB3 = button(TDS, "delete current heading",
             tdsF3)
    tdsB4 = button(TDS, "print table", tdsF4)
    tdsB5 = button(TDS, "shutdown server", tdsF5)
    close(TDS,true, closeCB)
}
// TDS required methods  -- no acknowledge necessary
void fromTds (string key, data) {
    ack "message from TDS (" key ", " data ")"
}
// all installed
ack "API Demo installed"
```

Starting the client

The DOORS/TDS client is started with the file `apistart.dxl`, which contains:

```
if (!TDSIsShowing){
     initTDS
}
TDSIsShowing = true
show TDS
```

Protocol

The protocol for the exchange of messages is as follows:

1. Rational DOORS starts the server process.
2. Rational DOORS sends a message to the TXL process running `server.txl` by calling `tdsSend`.
3. `server.txl` accepts the Rational DOORS client.
4. If the message is `quit_`, `tdsSend` takes no further action, and `server.txl` waits for the next client.
5. If the message is `shutdown_`, `tdsSend` takes no further action, and `server.txl` exits, causing the xterm or DOS shell to exit.
6. Any other message is executed by `server.txl` as a TXL program.
7. `tdsSend` then expects a reply.

8. If the reply is `errors_`, report an error.
9. If the reply is `done_`, stop.
10. If the reply is anything else, execute it as a DXL program and wait for a further message from `server.txl`.

This simple protocol allows either side to send code to the other for execution, but Rational DOORS must always be the initiator. This is the main characteristic of a Rational DOORS active link.

The server and client code can be reused, with minor modification, as the basis for any other tool server (Rational DOORS active link).

Rational DOORS passive link

A passive link is where the application, for example, TDS, wishes to drive Rational DOORS rather than act as a server. To do this, use the Rational DOORS C API's services to drive the DXL server or its batch emulation.

The programs `dxlips`, `dxlipf` and `dxlfile` (described in "Using the DXL server," on page 17) are examples of Rational DOORS passive link programs; their source is in `$DOORSHOME/api`. Rational DOORS can only reply to their messages with core DXL messages.

1. Run the file `api2init.dxl`, which is in `$DOORSHOME/lib/dxl/ example`.
2. To start the DXL server using TCP/IP sockets run the DXL shown in "Using the DXL server," on page 17.
3. From a shell, run `$DOORSHOME/bin/dxlips "reply"` for TCP/IP sockets.

This causes all the headings of the first formal module in the current project to be printed in the shell. The definition of reply in `api2init.dxl` is as follows:

```
/* Example function used to illustrate the
DXL server.
*/
void reply() {
    ack "reply"
    Object o
    string s
    Module mnull
    for s in current Project do {
        if (type module s  "Formal") {
            m = edit(s,false)
            break
        }
    }//for loop
    if (null m)
        ack "no formal modules"
```

```
        else
            for o in m do
                replyAPI "print \"" (o."Object
                            Heading" "") "\n\""
    }//reply
```

The `replyAPI` function sends a message back to `dxlips` to be executed as a core DXL program.

Source of dxlips.c

```
/* dxlips.c */
/*
 Copyright (c) 1993-2000 Telelogic AB.
 See Rational DOORS manuals for copying conditions.
 Copy this file to a different location before
 modifying it.
*/
/*
   Use TCP/IP sockets to connect to DXL server
   from DXL interaction window execute:
   evalTop_ "initDXLServer server 5093"
   to initialize server.
*/
#include <doors/api.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define DXLDEFPORTNO 5093
#define DXLDEFIPHOST "127.0.0.1"
extern char* getenv();
int main (argc, argv)
    int argc;
    char* argv[];
{
    char* portnos = getenv("DXLPORTNO");
    /* string of portno */
    char* host = getenv("DXLIPHOST");
    unsigned short portno;
    if (argc < 2)
        apiError("usage: dxlips \"message\"");
    if (portnos==NULL)
        portno = DXLDEFPORTNO;
    else
        portno = atoi(portnos);
    if (host==NULL)
        host = DXLDEFIPHOST;
    apiInitLibrary((char*)NULL,(char*)NULL,
                    (char*)NULL);
```

```
        printf("portno = %d, host = %s\n",
                 portno, host);
        apiConnectSock(portno, host);
        apiSend(argv[1]);
        if (apiErrorState == DOORS_API_OK &&
             strcmp(argv[1],"shutdown_")!=0)
            apiSend("quit_");
        apiFinishLibrary();
        return 0;
    }
    /* end of dxlips.c */
```

It is part of `apiSend`'s job to wait to see whether Rational DOORS uses `replyAPI`. Refer to "DXL API integration features," on page 21 for more information.

Registering methods

You can make a more useful passive link by registering methods with the Rational DOORS API, which can be executed by the reply. This allows a passive link program to retrieve data from Rational DOORS and manipulate it. The program `activeIP.c` is the same as `dxlips.c` except that it has the definition:

```
BEGIN_FN(myRepFn,1,0)
    P_(char*,r);
    END_DECLS;
    printf("\"%s\" and again \"%s\"\n", r, r);
END_FN
```

and the line:

```
apiInstall("void myReply(string)", myRepFn);
```

The file `api2init.dxl` also defines the function `reply2`, which is the same as `reply` except for the line:

```
replyAPI "myReply \"" (o."Object Heading") "\"".
```

Execute:

```
activeIP "reply2"
```

to see that data from Rational DOORS can be extracted and manipulated by the Rational DOORS passive link program `activeIP`.

The `replyAPI` function can present a possible hazard to Rational DOORS if the client side of the DXL server is not expecting a reply. For example, `replyAPI` could have been accidentally executed when there is no client currently connected. For this reason, a time limit of 20 seconds is given for the client to respond; this time limit can be changed with the `setAPIClientTimeout` function.

The protocol between clients, the DXL server and Rational DOORS is robust against errors in any of the messages. The `reply3` function, installed by `api2init.dxl`, deliberately returns a bad message to the DXL client, which recovers from the error and prints a message, as does Rational DOORS. To see the effect, try the following:

```
activeIP "reply3"
```

The `reply4` function, installed by `api2init.dxl`, causes `dxlips` to execute a small script that prints today's date. To see the effect, try the following:

```
dxlips "reply4"
```

All Rational DOORS passive links should follow closely the example set in this section. The interface should consist of a well-defined set of commands implemented as DXL functions that are then called by the external tool via `dxlips` (or a similar program). This minimizes the traffic through the IPC channel and will lead to a cleaner interface between the tools.

Working with OLE objects

Rational DOORS supports OLE from DXL both as an automation server which can implement a Rational DOORS passive link, and as an OLE client which can implement a Rational DOORS active link.

Refer to the *DXL Reference Manual* for details of these DXL features.

For further information, reference Rational DOORS's Microsoft Office import and export tools, which provide code examples of these features, and can be found using the Library option in the DXL Interaction window.

Listing of tds.c

```
/*
 Copyright (c) 1993-2000 Telelogic AB.
 See Rational DOORS manuals for copying conditions.
 Copy this file to a different location before
 modifying it.
*/
/* This module implements a DXL-like language for
   TDS.
   TDS (Toy Database System) serves as an example
   of how to integrate external tools with Rational DOORS.
*/
#include <doors/api.h>      /* API services */
#include "tds.h"   /* this file's entry points */
#include "tdsfns.h"         /* the TDS API */
/* start declaring TDS API driven functions */
```

```
BEGIN_FN(tdsCreateFn,1,1)
    P_(char*,name);
    Table* tab;
    END_DECLS;
    tab = tdsCreate(name);
    RETURN_(tab);  /* return the created table */
END_FN
BEGIN_FN(tdsEntryFn,2,1)
    P_(Table*,tab);
    P_(char*,key);
    Entry* e;
    END_DECLS;
    e = tdsEntry(tab,key,TRUE);
    RETURN_(e);
END_FN
BEGIN_FN(tdsPutFn,2,0)
    P_(Entry*, e);
    P_(char*,data);
    END_DECLS;
    tdsPut(e,data);
END_FN
BEGIN_FN(tdsGetFn,2,1)
    P_(Table*,tab);
    P_(char*,key);
    char* data;
    END_DECLS;
    data = tdsGet(tab, key);
    RETURN_(data);
END_FN
BEGIN_FN(tdsGetKeyFn,1,1)
    P_(Entry*, e);
    END_DECLS;
    RETURN_(e->key);
END_FN
BEGIN_FN(tdsGetDataFn,1,1)
    P_(Entry*, e);
    END_DECLS;
    RETURN_(e->data);
END_FN
BEGIN_FN(tdsGetNameFn,1,1)
    P_(Table*, t);
    END_DECLS;
    RETURN_(t->name);
END_FN
BEGIN_FN(tdsDeleteEntryFn,2,1)
    P_(Table*,tab);
    P_(char*,key);
    int status;
    END_DECLS;
```

```
    status = tdsDeleteEntry(tab,key);
    RETURN_(status);
END_FN
BEGIN_FN(tdsDeleteTableFn,1,1)
    P_(Table*,tab);
    int status;
    END_DECLS;
    status = tdsDeleteTable(tab);
    RETURN_(status);
END_FN
BEGIN_FN(tdsInfoFn,0,0)
    END_DECLS;
    tdsInfo();
END_FN
BEGIN_FOR_DO(tdsDoFn,Table*,tab,Entry*,scan)
   tdsDo(tab,scan)
       PROCESS_DO(scan);
END_FOR_DO
BEGIN_FOR_DO(tdsDoAllFn,Table*,tab,Table*,scan)
   tdsTabDo(scan)
       PROCESS_DO(scan);
END_FOR_DO
/***********************************************
 ** tdsInitAPI
*/
global void tdsInitAPI(void)
{
    tdsInit();
    /* Declare the XTC types for TDS */
    apiParse("struct Table {};
              struct Entry {};
               Table All=null;");
    /* Declare Status constants */
    apiParse("const int StatusOK = addr_(%d) ;",
              StatusOK);
    apiParse("const int StatusBadDeleteEntry =
              addr_(%d) ;", StatusBadDeleteEntry);
    apiParse("const int StatusBadDeleteTable =
              addr_(%d) ;", StatusBadDeleteTable);
    /* Declare the API entry points */
    apiInstall("Table create (string)",
                 tdsCreateFn);
    apiInstall("Entry ::[] (Table, string)",
                 tdsEntryFn);
    apiInstall("void ::= (Entry, string)",
                 tdsPutFn);
    apiInstall("string ::[] (Table,string)",
                 tdsGetFn);
```

```
    apiInstall("string key (Entry)",
                 tdsGetKeyFn);
    apiInstall("string data (Entry)",
                 tdsGetDataFn);
    apiInstall("string ::* (Entry)",
                 tdsGetDataFn);
    apiInstall("string name (Table)",
                 tdsGetNameFn);
    apiInstall("int delete (Table,string)",
                 tdsDeleteEntryFn);
    apiInstall("int delete (Table)",
                 tdsDeleteTableFn);
    apiInstall("void info ()", tdsInfoFn);
    apiInstall("void ::do (Entry&, Table, void)",
                 tdsDoFn);
    apiInstall("void ::do (Table&, Table, void)",
                 tdsDoAllFn);
}  /* tdsInitAPI */
/*
  tdsFinishAPI
*/
global void tdsFinishAPI(void)
{
    tdsFinish();
}  /* tdsFinishAPI */
```

7 Contacting support

This chapter contains the following topics:

- Contacting IBM Rational Software Support
- Prerequisites
- Submitting problems
- Other information

Contacting IBM Rational Software Support

If the self-help resources have not provided a resolution to your problem, you can contact IBM Rational Software Support for assistance in resolving product issues.

Note If you are a heritage Telelogic customer, you can go to http://support.telelogic.com/toolbar and download the IBM Rational Telelogic Software Support browser toolbar. This toolbar helps simplify the transition to the IBM Rational Telelogic product online resources. Also, a single reference site for all IBM Rational Telelogic support resources is located at http://www.ibm.com/software/rational/support/telelogic/

Prerequisites

To submit your problem to IBM Rational Software Support, you must have an active Passport Advantage® software maintenance agreement. Passport Advantage is the IBM comprehensive software licensing and software maintenance (product upgrades and technical support) offering. You can enroll online in Passport Advantage from http://www.ibm.com/software/lotus/passportadvantage/howtoenroll.html.

- To learn more about Passport Advantage, visit the Passport Advantage FAQs at http://www.ibm.com/software/lotus/passportadvantage/brochures_faqs_quickguides.html.
- For further assistance, contact your IBM representative.

To submit your problem online (from the IBM Web site) to IBM Rational Software Support, you must additionally:

- Be a registered user on the IBM Rational Software Support Web site. For details about registering, go to http://www-01.ibm.com/software/support/.
- Be listed as an authorized caller in the service request tool.

Submitting problems

To submit your problem to IBM Rational Software Support:

1. Determine the business impact of your problem. When you report a problem to IBM, you are asked to supply a severity level. Therefore, you need to understand and assess the business impact of the problem that you are reporting.

Use the following table to determine the severity level.

Severity	Description
1	The problem has a *critical* business impact: You are unable to use the program, resulting in a critical impact on operations. This condition requires an immediate solution.
2	This problem has a *significant* business impact: The program is usable, but it is severely limited.
3	The problem has *some* business impact: The program is usable, but less significant features (not critical to operations) are unavailable.
4	The problem has *minimal* business impact: The problem causes little impact on operations or a reasonable circumvention to the problem was implemented.

2. Describe your problem and gather background information, When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Rational Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these questions:

- What software versions were you running when the problem occurred?

  To determine the exact product name and version, use the option applicable to you:

    - Start the IBM Installation Manager and select **File** > **View Installed Packages**. Expand a package group and select a package to see the package name and version number.
    - Start your product, and click **Help** > **About** to see the offering name and version number.
- What is your operating system and version number (including any service packs or patches)?
- Do you have logs, traces, and messages that are related to the problem symptoms?
- Can you recreate the problem? If so, what steps do you perform to recreate the problem?
- Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, or other system components?
- Are you currently using a workaround for the problem? If so, be prepared to describe the workaround when you report the problem.

3. Submit your problem to IBM Rational Software Support. You can submit your problem to IBM Rational Software Support in the following ways:

- **Online**: Go to the IBM Rational Software Support Web site at https://www.ibm.com/software/rational/support/ and in the Rational support task navigator, click **Open Service Request.** Select the electronic problem reporting tool, and open a Problem Management Record (PMR), describing the problem accurately in your own words.

  For more information about opening a service request, go to http://www.ibm.com/software/support/help.html

  You can also open an online service request using the IBM Support Assistant. For more information, go to http://www-01.ibm.com/software/support/isa/faq.html.
- **By phone**: For the phone number to call in your country or region, go to the IBM directory of worldwide contacts at http://www.ibm.com/planetwide/ and click the name of your country or geographic region.
- **Through your IBM Representative**: If you cannot access IBM Rational Software Support online or by phone, contact your IBM Representative. If necessary, your IBM Representative can open a service request for you. You can find complete contact information for each country at http://www.ibm.com/planetwide/.

If the problem you submit is for a software defect or for missing or inaccurate documentation, IBM Rational Software Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Rational Software Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the IBM Rational Software Support Web site daily, so that other users who experience the same problem can benefit from the same resolution.

Other information

For Rational software product news, events, and other information, visit the IBM Rational Software Web site on http://www.ibm.com/software/rational/.

8 Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,

THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
1 Rogers Street
Cambridge, Massachusetts 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Additional legal notices are described in the legal_information.html file that is included in your software installation.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.html.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.