

Conga User Guide

Conga version **3.1**



DYALOC

The tool of thought for software solutions

*Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2018 by Dyalog Limited
All rights reserved.*

Conga User Guide

Conga version 3.1
Document Revision: 20180615310

No part of this publication may be reproduced in any form by any means without the prior written permission of Dyalog Limited.

Dyalog Limited makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Dyalog Limited reserves the right to revise this publication without notification.

*email: support@dyalog.com
<https://www.dyalog.com>*

TRADEMARKS:

SQAPL is copyright of Insight Systems ApS.

Array Editor is copyright of davidliebtag.com

Raspberry Pi is a trademark of the Raspberry Pi Foundation.

Oracle®, Javascript™ and Java™ are registered trademarks of Oracle and/or its affiliates.

UNIX® is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Windows® is a registered trademark of Microsoft Corporation in the United States and other countries.

macOS® and OS X® (operating system software) are trademarks of Apple Inc., registered in the U.S. and other countries.

All other trademarks and copyrights are acknowledged.

Contents

Preface	iv
1 About This Document	1
1.1 Audience	1
1.2 Conventions	1
2 Introduction	3
3 Installation	4
3.1 Compatibility	4
3.2 Initialisation	4
4 Getting Started	6
4.1 Conga Objects	6
4.1.1 Conga Object Types	6
4.1.2 Conga Object States	8
4.1.3 Conga Object Modes	10
4.2 A Simple Conga Client	13
4.3 A Simple Conga Server	14
4.4 Command Mode	15
4.5 Parallel Commands	17
4.5.1 Multi-threading	19
4.6 Deflate HTTP Compression	20
4.6.1 How HTTP Compression Works	20
4.6.2 Deflate Compression	21
5 Secure Connections	23
5.1 CA Certificates	24
5.2 Client and Server Certificates	25
5.2.1 Certificate Stores	26
5.2.2 Revocation Lists	26
5.3 Creating a Secure Client	26
5.4 Creating a Secure Server	28
5.5 Using the DRC.X509Cert Class	29
5.5.1 Certificate Chains	31
6 The Conga Workspace	33
6.1 Namespace: Samples	34
6.1.1 Function: Samples.Test*	34
6.2 Namespace: WebServer	35
6.2.1 Function: WebServer.Run	35
6.3 Namespace: RPCServer	36

6.3.1	Function: RPCServer.Run	37
6.4	Class: FTPClient	38
6.5	Namespace: TODServer	39
6.5.1	Function: TODServer.Run	39
A	Technical Reference	41
A.1	DRC Return Codes	41
A.2	Function: DRC.Certs	42
A.3	Function: DRC.ClientAuth	42
A.4	Function: DRC.Close	43
A.5	Function: DRC.Clt	43
A.6	Function: DRC.Describe	46
A.7	Function: DRC.Error	47
A.8	Function: DRC.Exists	47
A.9	Method: DRC.Flate.Deflate	48
A.10	Method: DRC.Flate.Inflate	48
A.11	Method: DRC.Flate.IsAvailable	49
A.12	Function: DRC.GetProp	49
A.13	Function: DRC.Init	52
A.14	Function: DRC.Names	53
A.15	Function: DRC.Progress	54
A.16	Function: DRC.Respond	54
A.17	Function: DRC.Send	55
A.18	Function: DRC.ServerAuth	57
A.19	Function: DRC.SetProp	57
A.20	Function: DRC.Srv	59
A.21	Function: DRC.Tree	61
A.22	Function: DRC.Version	63
A.23	Function: DRC.Wait	63
A.24	Class: DRC.X509Cert	65
A.24.1	Instances of the DRC.X509Cert Class	67
A.25	Operator: Samples.HTTPCmd	69
A.26	Function: Samples.HTTPGet	71
A.27	Function: Samples.TestFTPClient	73
A.28	Function: Samples.TestSecureWebClient	73
A.29	Function: Samples.TestWebClient	75
A.30	Function: WebServer.Run	76
B	Certificates	77
B.1	PEM File Format	77
B.2	Generating Certificates and Keys	77
C	TLS Flags	81

D	Conga Libraries	83
E	Error Codes	85
F	Change History	87
	F.1 Version 3.1	87
	F.2 Version 3.0	87
	F.3 Version 2.7	88
	F.4 Version 2.6	88
	F.5 Version 2.5	89
	F.6 Version 2.4	89
	F.7 Version 2.3	89
	F.8 Version 2.2	90
	F.9 Version 2.1	90
	Index	92

Preface

This document is accurate for Conga version 2.7, released with Dyalog version 15.0. Enhancements for Conga version 3.0 (released with Dyalog version 16.0) and Conga version 3.1 (released with Dyalog version 17.0) are currently detailed in a [separate document](#), the *Conga User Guide – Supplement*; the material in supplement will be incorporated into a future revision of this document.

Conga versions 3.0 and 3.1 are designed to be upwards compatible with Conga version 2.7 and earlier releases. This document accurately describes the use of Conga version 3.1 except that a number of samples have been removed from the distributed workspace **conga.dws** and new features are not included. The following components have been removed from the distributed **conga.dws**:

- The TODServer example has been completely retired.
- The FTPClient, along with many of the utilities that used to be found in the `HttpUtils` and `Samples` namespaces, have been moved to new locations (described in the *Code Libraries Reference Guide*).
- The RPCServer and WebServer examples have been replaced by new code that uses new features of Conga version 3.0; these can be found in the **[DYALOG]/Samples/Conga** directory and are described in the *Conga User Guide – Supplement*.

1 About This Document

This document is a complete guide to Conga, Dyalog's framework for TCP/IP communications. It describes the tools with which Conga can be used to create a variety of clients and servers using protocols based on TCP/IP, including HTTP, HTTPS, FTP, Telnet and SMTP. It covers Conga support for secure communications (using SSL/TLS) and communication between APL processes (allowing them to exchange native APL data directly). It also introduces the Conga workspace, which includes a comprehensive collection of samples showing the implementation of various types of servers and clients, and contains a technical reference of the namespaces, classes and functions provided with Conga.

1.1 Audience

It is assumed that the reader has a reasonable understanding of Dyalog and server client connection protocols; a working knowledge of HTTP/FTP/SMTP is needed to understand the samples provided with the **conga** workspace.

For information on the resources available to help develop your Dyalog knowledge, see <http://www.dyalog.com/introduction.htm>.

1.2 Conventions

Unless explicitly stated otherwise, all examples in Dyalog documentation assume that `⎕IO` and `⎕ML` are both 1.

Various icons are used in this document to emphasise specific material.

General note icons, and the type of material that they are used to emphasise, include:



Hints, tips, best practice and recommendations from Dyalog Ltd.



Information note highlighting material of particular significance or relevance.



Legacy information pertaining to behaviour in earlier releases of Dyalog or to functionality that still exists but has been superseded and is no longer recommended.



Warnings about actions that can impact the behaviour of Dyalog or have unforeseen consequences.

Although the Dyalog programming language is identical on all platforms, differences do exist in the way some functionality is implemented and in the tools and interfaces that are available. A full list of the platforms on which Dyalog version 17.0 is supported is available at www.dyalog.com/dyalog/current-platforms.htm. Within this document, differences in behaviour between operating systems are identified with the following icons (representing macOS, Linux, UNIX and Microsoft Windows respectively):



2 Introduction

Conga (also known as the Dyalog Remote Communicator) is a tool for communication between applications. It can transmit APL arrays between two Dyalog applications that both use Conga (that is, both call functions within the DRC namespace), and it can exchange messages with many other applications, for example, HTTP servers (also known as web servers), web browsers and other web clients/servers including Telnet, SMTP and POP3.

Uses of Conga include, but are not limited to, the following:

- Retrieving information from – or uploading data to – the internet.
- Accessing internet-based services like FTP, SMTP or Telnet.
- Writing an APL application that acts as a Web (HTTP) Server, mail server or any other kind of service available over an intranet or the internet.
- Implementing APL Remote Procedure Call (RPC) servers; these receive APL arrays from client applications, process data and return APL arrays as the result.

Conga supports secure communication using TLS (Transport Layer Security), which is the successor to SSL (Secure Sockets Layer). Conga makes it easy for APL developers to embed client or server components in APL applications and simplifies the process of making remote calls in a multi threaded client environment.

Although Conga currently only uses the TCP protocol, other communication mechanisms could be added in the future. Conga hides many of the details of TCP socket handling and notifies the application of incoming data, connection events and errors so that all the application needs to do is handle the data that arrives. Dyalog Ltd recommends Conga as the mechanism for handling TCP-based communications in preference to the now-outdated TCPSocket object.

Conga is used in many Dyalog tools including MiServer (Dyalog's APL-based web server), SAWS (the Stand-Alone Web Service framework), the Dyalog File Server and isolates.



If you redistribute code that uses Conga, please see the *Licences for third-party components* document in the **[DYALOG]/Help** directory of your installation.

3 Installation

Conga is implemented as a Microsoft Windows Dynamic Link Library or a UNIX/Linux Shared Library. The library is loaded and accessed through the DRC namespace, which is part of the `conga` workspace; the `conga` workspace also contains a number of sample applications.

No installation is required – Conga is supplied with Dyalog.

3.1 Compatibility

All versions of Conga from version 2.0 onwards are:

- compatible with all supported Dyalog versions.
- compatible with each other.

When the server and client are both Conga objects, they:

- do not have to be running the same version of Conga.
- do not have to be running the same version of Dyalog.

However, standard Dyalog interoperability rules apply – any specific functionality that is required (in Conga or Dyalog) must be available at both ends of the connection. For details of the changes made in each Conga version, see *Appendix F*.

For compression to work, the client and server both need to support the same compression scheme (see *Section 4.6*).

3.2 Initialisation

Before using any of the functions in the DRC namespace, the system needs to be initialised by loading the library (Microsoft Windows DLL or UNIX Shared Library).

To initialise the system

1. Access the DRC namespace in the appropriate way:

- If you are experimenting with functionality before adding Conga to an application, then load the `conga` workspace:

```
)LOAD conga
...\ws\conga.dws saved Fri Jul 24 17:21:04 2015
```

- If you are writing an application whose behaviour should remain unchanged irrespective of future changes to Conga, then copy the DRC namespace from the `conga` workspace into the application's workspace:

```
)COPY conga DRC
...\ws\conga.dws saved Fri Jul 24 17:21:04 2015
```



If an application is shipped that includes Conga, then the relevant libraries will also need to be shipped. For more information, see *Appendix D*.

2. Initialise the library:

```
DRC.Init ''
0 Conga loaded from: ...\conga27x64Uni
```

The examples throughout this document assume that the system has been initialised, that is, the above steps have already been followed.

4 Getting Started

This chapter introduces Conga client and server objects and demonstrates their use through simple examples.

The purpose and syntax of the functions and methods used in this chapter are detailed in *Appendix A*.

4.1 Conga Objects

A *Conga object* is a named object created inside Conga but outside the workspace. Each Conga object has specific properties that can be queried and/or set (some properties are read-only).

Conga object names cannot exceed 32 characters in length and must not contain null characters.

4.1.1 Conga Object Types

There are six possible *types* of Conga object – these are listed with their identifying object type code in *Table 4-1* (object type codes are used by several of the DRC.* functions in Conga – see *Appendix A*).

Table 4-1: Conga object types

Code	Object Type	Description
0	Root	The highest level object. The root object contains information about the Conga installation; it is created with the DRC. <i>Init</i> function.

Table 4-1: Conga object types (continued)

Code	Object Type	Description
1	Server*	A server object listens for connections from clients – the client can be any TCP/IP client and does not have to be a Conga object. It also receives, processes and responds to requests from connections.
2	Client*	A client object connects to a server, sends requests to it and receives responses from it; the server can be any TCP/IP server and does not have to be a Conga object.
3	Connection	A server object can respond to multiple client connections; a connection object maintains information pertaining to each client connection.
4	Command	A command object represents an individual request (from a client) or response (from a server). Command objects only exist for servers and clients in <i>Command</i> mode (see <i>Section 4.1.3</i>).
5	Message	Message objects are created by servers using the <code>DRC.Progress</code> function and sent to client objects. Message objects only exist for servers and clients in <i>Command</i> mode (see <i>Section 4.1.3</i>).

* A client is said to be *using Conga* if it is communicating through a client instance that was set up using the `DRC.Client` function (see *Section A.5*); a server is said to be using Conga if it is communicating through a server instance that was set up using the `DRC.Server` function (see *Section A.20*). The other end of the connection can also be using Conga or it can be a non-Conga object that understands TCP/IP (for example, a web browser or web server).

The ERD (Entity-Relationship Diagram) in *Figure 4-1* shows how servers and clients are related to each other and to other Conga objects. At least one of the server side or client side must be using Conga.

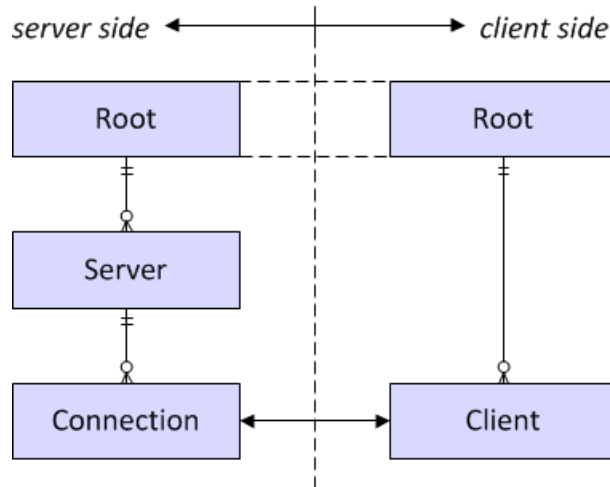


Figure 4-1: Conga object ERD (standard crow's foot notation)

Communication between a server's connection object and a client object depend on the connection mode (see [Section 4.1.3](#)). The ERD in [Figure 4-2](#) shows how the remaining two Conga object types relate to the connection and client objects – this is only valid when both the server and client are in *Command* mode. The sequence of events starts when the client sends a command through the connection to the server; optionally, the server sends messages through the connection to the client before sending a final response to the command.

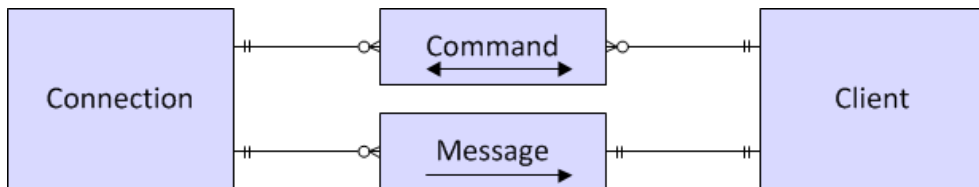


Figure 4-2: Communication ERD in Command mode (standard crow's foot notation)

For more information on connection modes, see [Section 4.1.3](#) and [Section 4.4](#).

4.1.2 Conga Object States

Each Conga object has a *state*, that is, temporary condition in which it exists as it progresses through its cycle from creation to deletion. The possible states and the Conga object types ([Section 4.1.1](#)) that can exist in each of these states are detailed in [Table 4-2](#).

Table 4-2: Conga object states and the object types that can exist in those states

Code	Object State	Conga Object Types						Description
		0	1	2	3	4	5	
0	New	x	x	x	x	x	x	Transient state that the object exists in after it is created but before it has been initialised.
1	Incoming				x			A connection has been established but a <code>Connect</code> event has not yet been received.
2	RootInit	x						The root object exists and is connected to the Conga library (Microsoft Windows DLL or UNIX/Linux Shared Library).
3	Listen		x					The server is listening for incoming connection attempts.
4	Connected			x	x			The client/connection is connected to its connection/client peer.
5	APL					x	x	The thread that handles socket communications has a full buffer and no further processing can occur until the application calls the <code>DRC.Wait</code> function.
6	ReadyToSend					x	x	Data is ready to be sent.
7	Sending					x	x	Sending data.
8	Processing					x		The command has been passed to the server but no response has been issued.
9	ReadyToRecv					x	x	Waiting for data.
10	Receiving					x	x	Receiving data.

Table 4-2: Conga object states and the object types that can exist in those states
(continued)

Code	Object State	Conga Object Types						Description
		0	1	2	3	4	5	
11	Finished	x	x	x	x	x	x	All data has been transmitted/received, connections closed and commands finished.
12	MarkedForDeletion	x	x	x	x	x	x	The Conga object is ready for deletion.
13	Error		x	x	x	x	x	An error has occurred.
14	<i>internal</i>	-	-	-	-	-	-	<i>internal</i>
15	<i>internal</i>	-	-	-	-	-	-	<i>internal</i>
16	SocketClosed			x	x			The socket has been closed.
17	APLLast			x	x			The connection has been closed but uncollected data still exists in the thread that handles socket communications.
18	SSL			x	x			The client/connection is negotiating an SSL connection with its connection/client peer.

4.1.3 Conga Object Modes

Conga clients and servers support five different modes for connection, that is, formats in which data can be transmitted:

- *Text*

Allows transmission of character strings. Character strings can only comprise characters with Unicode code points less than 256. To transmit characters outside this range, Dyalog Ltd recommends that you either use UTF-8 character encoding (for information on this, see [Unicode](#) in the *Dyalog APL Language Reference Guide*) or switch to *Raw* mode and convert the character string to the appropriate format (for example, by applying [Unicode](#)).

- *BlkText*

As *Text* mode, but each data transmission is considered as a block. If a block exceeds the maximum size (as defined by the `BufferSize` parameter of the `DRC.Cl t/DRC.Srv` function) then the peer object receiving the transmission can reject it. Each block includes a header stating the:

- block length: 32-bit integer giving the exact length of the block. Determined by the network/operating system.
- magic number (optional): 32-bit integer used to check that the data has not been corrupted. Set using the `DRC.Cl t/DRC.Srv` function (the `Magic` parameter).

Each block is assigned the event name `Block`. If the connection closes before all blocks have been processed, then the final block to be processed is assigned the event name `BlockLast`. The third element returned by the `DRC.Wait` function indicates the event name (see *Section A.23*).

Unless explicitly specified otherwise, information about *Text* mode can be assumed to apply to *BlkText* mode too. Only valid when both the server and client are using Conga.

- *Raw*

Similar to *Text* mode, except that data is represented as integers in the range 0 to 255 (for coding simplicity, negative integers -128 to -1 are also accepted and mapped to 128-255).

- *BlkRaw*

As *Raw* mode, but each data transmission that exceeds the maximum size (as defined by the `BufferSize` parameter of the `DRC.Cl t/DRC.Srv` function) is chunked into blocks. Each block includes a header stating the:

- block length: 32-bit integer giving the exact length of the block. Determined by the network/operating system and cannot exceed the `BufferSize`.
- magic number: 32-bit integer unique to the blocks in a single data transmission; used to identify blocks in the same transmission. Set using the `DRC.Cl t/DRC.Srv` function (the `Magic` parameter).

If data is too large to fit into a single block, then multiple blocks are created; each block is assigned the event name `Block`. If the connection closes before all blocks have been processed, then the final block in the transmission is assigned the event name `BlockLast`. The third element returned by the `DRC.Wait` function indicates the event name (see *Section A.23*).

Unless explicitly specified otherwise, information about *Raw* mode can be assumed to apply to *BlkRaw* mode too. Only valid when both the server and client are using Conga.

- *Command*

Each transmission is a complete APL object in a binary format. This is the default mode. Only valid when both the server and client are using Conga. For more information on *Command* mode, see *Section 4.4*.

A client and server can only exchange data if they are running in compatible modes. Specifically, a *Command* mode client must be connected to a *Command* mode server and a *Text* mode or *Raw* mode client must be connected to a *Text* mode or *Raw* mode server; *BlkText* mode and *BlkRaw* mode are interchangeable with *Text* mode and *Raw* mode as long as the requisite header (containing the field length and magic number) is added.

Text mode and *Raw* mode are typically used when only one end of the connection is an APL application.

Command mode is the optimal way for APL clients and servers to communicate with each other, because:

- the internal representation is the binary format used by APL; this is more compact than a textual representation.
- numbers can be transmitted without having to be formatted and interpreted.
- no buffer size needs to be declared.

In *Command* mode, *BlkText* mode or *BlkRaw* mode, each transmission comprises an entire APL array or block of data; the `DRC.Wait` function does not report incoming data until the entire APL array or block of data has arrived. In *Text* mode and *Raw* mode, byte streams are transmitted – in *Text* mode these are translated to a character vector on receipt, in *Raw* mode, integers between 0 and 255 are returned; the `DRC.Wait` function reports incoming data each time a TCP packet arrives or when the receive buffer is full. The recipient may need to buffer incoming data in the workspace and analyse it to determine whether a complete message has arrived.

In *Text* and *Raw* modes, an EOM termination string can be set. In this situation, the `DRC.Wait` function terminates on receipt of the specified termination string. If an empty termination string is specified, then the `DRC.Wait` function terminates when the buffer contains `BufferSize` bytes (see *Section A.5* and *Section A.20*). If an EOM termination string is not specified, then the `DRC.Wait` function returns data each time a TCP packet is received. If a TCP packet is larger than `BufferSize` bytes then the data is returned in blocks of `BufferSize` bytes.

4.2 A Simple Conga Client

A Conga *client* establishes contact with a service that is already running and listening on a pre-determined port at a known TCP address. The service could be an APL application that has created a Conga server or it could be any application or service that provides services through TCP sockets. For example, most UNIX systems (and many Microsoft Windows servers) provide a set of simple services like a Time of Day (TOD) service or a Quote of the Day (QOTD) service, both of which respond with a text message as soon as a connection is made to them; once the message has been sent, they immediately close the connection.

The function `DRC.Clt` can be used to create a Conga client. In the following example, this function is called with five elements in its right argument:

- the name to be used for the client object (`C1`)
- the IP address or name of the server machine providing the service (`localhost`)
- the port on which the service is listening (13 – the TOD service)
- the type of socket (`Text`)
- the size (in bytes) of the buffer that should be created to receive data (1000)

```
DRC.Clt 'C1' 'localhost' 13 'Text' 1000
1111 ERR_CONNECT_DATA /* Could not connect to host data port */
```

The error message that is generated follows the syntax for all error codes generated by functions in the DRC namespace (see *Section A.1*), that is, it is a vector in which:

- [1] is a return code (see *Section A.1*)
- [2] is the error name
- [3] is, optionally, additional information about the error.

The reason that this `DRC.Clt` function call failed with error code 1111 is that it was called on Microsoft Windows and Windows does not usually have a TOD service running. This issue can be resolved in any of the following ways:

- Enable the service on localhost. This can be done by going to **Control Panel > Programs and Features > Turn Windows features on or off** and selecting the **Simple TCP/IP services (i.e. echo, daytime etc)** checkbox. Reboot, then rerun the `DRC.Clt` function call:

```
DRC.Clt 'C1' 'localhost' 13 'Text' 1000
0 C1
```

The result code of zero indicates that the client was successfully created (any other code indicates failure).

- Call a different server machine that does provide a TOD service, for example:


```
DRC.Clt 'C1' 'myLinuxBox' 13 'Text' 0
0 C1
```

The result code of zero indicates that the client was successfully created (any other code indicates failure).

- Write a TOD service for localhost (see *Section 6.5*)

After the client object has been successfully created, incoming data can be received. Receiving data from the server involves calling the `DRC.Wait` function with the name of the client. For example:

```
]DISP DRC.Wait 'C1'
```



The returned message is a vector in which:

- [1] is the return code
- [2] is the object name
- [3] is the type of event (see A.23)
- [4] is data associated with the event

The client object can now be closed (good practice):

```
DRC.Close 'C1'
0
```

4.3 A Simple Conga Server

The TOD service referred to in *Section 4.2* is a very simple server and can be implemented by calling the `TODServer.Run` function in the conga workspace to create a server object (for the code of this function, see *Section 6.5.1*). The `TODServer.Run` function enters a loop where it waits for connections; this means that, to be able to experiment with using this service without starting a second APL session, it should be started using the *Spawn* operator (&) so that it runs in a separate thread:

```
TODServer.Run & 13
TOD Server started on port 13
```



For this to work on the Microsoft Windows operating system, the TOD service must be disabled (this can be done by going to **Control Panel > Programs and Features > Turn Windows features on or off** and unselecting the **Simple TCPIP services (i.e. echo, daytime etc)** checkbox, then rebooting).

The right argument in this function call is the port number; if a TOD service is already running on port 13, then an error message is returned and a different port must be used for the new service.

A client object can now be created, data received and the client object closed:

```
DRC.Clt 'C1' 'localhost' 13 'Text'
0 C1

DRC.Wait 'C1'
0 C1 Block 10:09:03 12-10-2015

DRC.Close 'C1'
0
```

The TOD server created by calling the `TODServer.Run` function is not restricted to only respond to Dyalog applications using Conga – it can be used by any program that is written to use a TOD service.

The server can be stopped as follows:

```
TODServer.DONE←1
TOD Server terminated.
```

4.4 Command Mode

Section 4.2 and *Section 4.3* used connections in *Text* mode, which are appropriate for most web applications. Even when remote procedure calls are made over the internet, with arguments and results containing arguments that are not simply text strings, the parameters are usually encoded using SOAP/XML, which is a text-based encoding.

The TOD server created by calling the `TODServer.Run` function in *Section 4.3* can be used by any program that is written to use a TOD service. It could be restricted to only respond to Dyalog applications using Conga by converting it to use *Command* mode – doing this means that it will return the time as a 7-element array in `⌈TS` format.

The conversion to *Command* mode is done by making the following changes to the `TODServer.Run` function code given in *Section 6.5.1*:

- [4] Remove 'Text' from the end of the line (*Command* is the default).
- Replace lines [13], [14] and [15] with the following:

```
[13]      :Case 'Connect'  A Ignore
[14]      :Case 'Receive'
[15]      {}##.DRC.Respond obj []TS
```

Unlike the *Text* mode TOD service, a server in *Command* mode cannot initiate the transmission of data when the connection is made, but can only respond to a request from a client.

In the above changes to the `TODServer.Run` function code, the `:Case 'Connect'` statement at line [13] does not have any associated code. However, code could be added here so that the TOD server could (for example) record connections. Without any code here, the TOD server responds with the current timestamp irrespective of the content of the request.

In *Text/Raw* mode, client and server can both initiate data transfer by calling the `DRC.Send` function (see *Section A.17*) – there is no concept of a request/respond protocol at the Conga level, although implementing an HTTP protocol over the connection can add such a protocol at the application level. However, in *Command* mode (and in *BlkRaw* mode and *BlkText* mode), Conga has an in-built protocol; communication on a connection is synchronous and consists of discrete commands. Each command comprises a request from the client followed by a response from the server; the server cannot initiate an unrequested transfer of data. The request message from the client and response message from the server are linked by an identifier (this is not the case in other modes). This means that, although the `DRC.Send` function can be used to send data from a client in *Command* mode, a different function must be used to send data from a server in *Command* mode – the `DRC.Respond` function (see *Section A.16*). The server can also call the `DRC.Progress` function (see *Section A.15*); this sends progress messages to the client while the server is processing a command, allowing the client to show the user a progress bar or other status information.

The modified TOD server can now be started. Ideally it should be started on a port other than port 13, so that it is not confused with a standard TOD server (if required, both the original and modified TOD servers could be run at the same time, in different threads):

```
TODServer.Run&913
TOD Server started on port 913
```

A *Command* mode Dyalog client of this TOD server can now be created and retrieve a numeric timestamp from the server:

```
DRC.Clt 'C1' 'localhost' 913
0 C1

DRC.Send 'C1' ''
0 C1.Auto00000000
```

The first element of the argument to the `DRC.Send` function can be either a client name or a connection name:

- if a client name is supplied (as in this example) then Conga generates a connection name and returns it as result element [2] in the format `clientname.connectionname` (in this example, `C1.Auto00000000`).
- if a connection name is supplied, then Conga returns it as result element [2].

```
DRC.Wait 'C1'
0 C1.Auto00000000 Receive 2015 10 19 9 36 48 845
```

Result element [4] is now a 7-element integer vector rather than a formatted timestamp; this is more performant on an APL client, but means that the TOD server is no longer usable by other TCP client programs that expect a *Text* mode TOD server.

Unlike the *Text* mode TOD server, the *Command* mode TOD server does not close the connection after sending a timestamp. This means that a second timestamp can be retrieved from the server (in this example the `DRC.Wait` function includes a maximum waiting time of 5 seconds):

```
DRC.Send 'C1' ''
0 C1.Auto00000001

DRC.Wait 'C1' 5000
0 C1.Auto00000001 Receive 2015 10 19 9 37 28 581
```

4.5 Parallel Commands

Although the *Command* mode protocol is synchronous, more than one command can be active at the same time – it is not necessary to wait for the response to one command before the next command is sent. In addition, multiple commands can be started and the results retrieved in any order.

EXAMPLE:

(In this example command names are specified, whereas in *Section 4.4* the command name was auto-generated)

```

    DRC.Send 'C1.TS1' ''
0  C1.TS1

    DRC.Send 'C1.TS2' ''
0  C1.TS2

    DRC.Wait 'C1.TS2' 1000
0  C1.TS2  Receive  2015 10 19 9 38 7 957

    DRC.Wait 'C1.TS1' 1000
0  C1.TS1  Receive  2015 10 19 9 37 57 965

```

The timestamps show that the TS1 command was executed before the TS2 command even though the results were retrieved in the reverse order.

The *Command* mode protocol allows multiple APL threads to work independently. A request message from the client and the associated response message from the server are linked by a command name; this means that any APL thread can wait for the result of a command, as long as it knows the command name. Multiple APL threads can share the same server connection; one APL thread can send a command and then dispatch a new APL thread to wait for and process the result of that command.

Command names can be reused as soon as the result has been received (but not before).

EXAMPLE:

Using client C1 and the modified TOD server created in *Section 4.4*:

```

    DRC.Send 'C1.TS1' ''
0  C1.TS1

    DRC.Send 'C1.TS2' ''
0  C1.TS2

    {[]TID,DRC.Wait w 1000}&" 'C1.TS1' 'C1.TS2'
2 0  C1.TS1  Receive  2015 11 16 11 25 28 850
3 0  C1.TS2  Receive  2015 11 16 11 25 32 474

```

This shows the asynchronous execution of a dynamic function; each of the two commands TS1 and TS2 calls the `DRC.Wait` function in a separate thread. Each function call returns the thread number and the result. Calls to the `DRC.Wait` function are thread switching points, which means that threads can be held while other threads continue execution.

4.5.1 Multi-threading

Conga supports multi-threaded applications; the ability to have a program work as both client and server simultaneously, without blocking other threads, has been an integral part of its design. All calls to Conga are implemented as asynchronous calls to an external library (Microsoft Windows DLL or UNIX/Linux Shared Library).

EXAMPLE:

The `RPCServer` namespace contains an example of a server working in *Command* mode (see *Section 6.3*) – this RPC server can execute APL statements in the server's workspace and return the results to client applications. Calling the `Samples.TestRPCServer` function starts the RPC server and spawns a number of APL threads; each APL thread makes a remote procedure call to a function in the `RPCServer` namespace. For each of the remote procedure calls, the server spawns a new APL thread; each of these new APL threads calls the `RPCServer.Process` function to handle the call to the function in the `RPCServer` namespace.



The status bar at the bottom of the **Session** window includes a field that displays the number of APL threads currently running (minimum value is 1). As the `Samples.TestRPCServer` function runs, the number of threads can be seen to increase before reverting to its initial value.

Additional web services, accessed through Conga, can be included by extending the `RPCServer.Process` function. These services can be external to the workspace or can run in the same workspace as everything else – in the latter situation each additional web service must be launched in a separate APL thread (as shown in *Section 4.3*). If the services are external to the workspace, then Conga uses multiple operating system threads to handle TCP communications; this is independent of the interpreter. Each result is returned to the APL thread that is waiting for it.

When developing an application, it is important to ensure that there is an APL thread waiting on each server object that has been created (otherwise requests will not be serviced). Having more than one APL thread waiting on the same object is not recommended – it can lead to unpredictable behaviour. For example:

```
Thread 1: DRC.Wait 'S1' 1000
Thread 2: DRC.Wait 'S1' 1000
```

could result in problems, whereas

```
Thread 1: DRC.Wait 'S1' 1000
Thread 2: DRC.Wait 'S2' 1000
```

is fine; this is determined by the application developer.



If a thread sustains an untrapped error then, by default, its execution is suspended and any other threads are paused; resuming execution of a suspended function only restarts the suspended thread. If the Session appears to lock while testing the multi-threading functionality, selecting the menu item **Threads > Resume all Threads** reactivates any paused threads.

4.6 Deflate HTTP Compression

Deflate is one of several content encoding schemes that can be used to implement HTTP compression; all major web browsers and web servers support deflate.

Although deflate can be used as a general data compression utility, its importance to Conga is its ability to provide HTTP compression. HTTP compression means that a smaller quantity of data needs to be transmitted across the network, thereby improving throughput between HTTP clients and servers. Typically a client will be a web browser (for example, Microsoft Internet Explorer, Google Chrome, Mozilla Firefox or Apple Safari) and a server will be a web server (for example, Microsoft IIS, Apache or IBM WebSphere). The **DRC.flate** class contains methods that provide support for deflate data compression, enabling Conga-based clients and servers to implement and support HTTP compression.



With a certain amount of adjusting from unsigned to signed integers, the `DRC.flate.Deflate` method produces the same result as calling `2(2191)`.

4.6.1 How HTTP Compression Works

For HTTP compression to work, the client and server both need to support the same compression scheme. They validate this in the following way:

1. The client informs the server of the compression schemes that it supports by including them as a comma-delimited list in the `Accept-Encoding` HTTP header that is sent with the request. The two predominant compression schemes currently are `gzip` and `deflate`.
2. The server receives the request from the client and examines the contents of the `Accept-Encoding` header. If the server also supports one of the compression schemes listed then it evaluates whether to encode the body of its response using that scheme. The server is not required to use any of the content encoding schemes in the `Accept-Encoding` header; if it does, then it informs the client which scheme it used in the `Content-Encoding` HTTP header that is sent with the response.
3. The client receives the response from the server and examines the contents of the `Content-Encoding` header (if found). It then decompresses the body of the response using that scheme.

4.6.2 Deflate Compression

Before implementing deflate compression, the mode of the clients and servers that will use deflate needs to be decided. The mode selected should be determined by the type and quantity of data to be compressed:

- *Text* mode is more convenient for the parsing and processing of the HTTP message wrapper.
- *Raw* mode is better suited for passing data to the `DRC.flate.Deflate` and `DRC.flate.Inflate` methods.

The open-source library that is used to implement deflate compression, `zlib`, prepends a 2-byte header (usually `120 156`), to the compressed data. Microsoft's Internet Explorer does not process these bytes correctly; the majority of web servers, therefore, tend to strip them off. This means that extra steps are required when implementing a client or server that uses deflate compression.

For a client that uses deflate compression:

1. In the request message, inform the server that deflate compression is supported by adding the following to the HTTP headers:
`Accept-Encoding: deflate`
2. When a response is received from the server:
 - i. check its HTTP headers for `Content-Encoding`. If this header is found and contains deflate, then prepend `120 156` to the response message before calling the `DRC.flate.Inflate` method.
 - ii. call the `DRC.flate.Inflate` method to decompress the data. This takes an integer vector of values in the range 0-255 as its right argument, which works well when the client is in *Raw* mode. However, if the client is in *Text* mode, then the call should be amended as follows to convert the data to a suitable form:
`'UTF-8' [] UCS DRC.flate.Inflate 256|83 [] DR data`

For a server that supports deflate compression:

1. In the request message from the client, check whether the following HTTP header is present:
`Accept-Encoding: deflate`
2. Assess whether HTTP compression is appropriate. For small responses, the CPU overhead to perform the compression could outweigh the gains of transmitting less data. In addition, data that is already in a compressed format (that is, files such as `.zip` and `.gz` files and graphics formats such as `.jpg/.jpeg` and `.gif`) is unlikely to be able to be compressed further.
3. If the header is present and compression is appropriate, then:
 - i. in the response message, inform the client that deflate compression has been used by adding the following to the HTTP headers:
`Accept-Encoding: deflate`

- ii. call the `DRC.flate.Deflate` method to compress the data. This takes an integer vector of values in the range 0-255 as its right argument, so works well when the server is in *Raw* mode. However, some data (such as web pages) are sent more efficiently in *Text* mode – this means that some conversion needs to be performed.

The following code is a modified sample from MiServer, Dyalog's APL based web server(see <https://www.github.com/Dyalog/MiServer>). It shows how the 2-byte header (120 156) is dropped from the result:

```
▽ (rc raw)←Compress buf;toutf8
:Implements Method ContentEncoder.Compress
toutf8←{3=10|⎕DR ω: 256|ω ⋄ 'UTF-8' ⎕UCS ω}
:Trap 0
  ⠇↓ drop of 789C header (IE cannot process it)
  raw←{(2×120 156≡2↑ω)↓ω}#.DRC.flate.Deflate toutf8 buf
  rc←0
:Else
  (rc raw)←1 ⎕DM
:EndTrap
▽
```

5 Secure Connections

Conga supports secure connections using SSL/TLS protocols. Secure connections allow client and server applications to:

- verify the identity of the partner that they are connected to.
- encrypt messages so that the contents cannot be deciphered by a third party, even when using text or raw mode connections.
- ensure that messages have not been tampered with by a third party during transmission.

SSL/TLS is a generic term for a set of related protocols used to add confidentiality and authentication to communications channels such as sockets. TLS (Transport Layer Security) is the successor to SSL (Secure Socket Layer) and is defined by the IETF and described in RFC 2246. There are only minor differences between the two protocols, so their names are often used interchangeably.

Recommended resources:

- [http://technet.microsoft.com/en-us/library/cc784450\(W5.10\).aspx](http://technet.microsoft.com/en-us/library/cc784450(W5.10).aspx) discusses the history, differences, benefits, etc. of SSL/TLS
- http://developer.mozilla.org/en/docs/Introduction_to_Public-Key_Cryptography provides an overview of the public key cryptography techniques used in SSL/TLS; the sections on the SSL protocol and CA (certificate authority) certificates are relevant for anyone who would like to make use of secure communications.
- <http://en.wikipedia.org/wiki/X.509> includes an introduction to how X.509 certificates and certificate authorities are used to establish trust.

To use SSL/TLS, Conga needs to be passed the necessary certificate and public key files when client and server objects are created.

Once a secure connection has been established, the same functions/methods are used to send and receive data (and with the same arguments) as when using a non secure connection.

5.1 CA Certificates

A *certificate authority* (CA) is a third party who is trusted by the parties at each end of a secure communication. The CA certifies that the named issuer of a certificate is the owner of the public key included within that certificate; the party at one end of a secure communication can then verify the identity of the party at the other end (known as the *peer*) using their certificate.

Verifying a CA's signature on a certificate requires having access to the CA's public certificate (often called a *root certificate*). Conga can be used to secure many different types of system, and can require multiple (and sometimes private) root certificates from several CAs.

All public root certificates that are located and downloaded for use with Conga should be placed in a single root certificate directory. The `DRC.SetProp` function can be used to inform Conga of the location of this directory. For example, the sample root certificates supplied with Conga can be used by entering:

```
DRC.SetProp '.' 'RootCertDir' (Samples.CertPath, 'ca')
0
```

(The `Samples.CertPath` function returns the location of the **TestCertificates** folder.)

Table 5-1 lists the download pages for root certificates for the most widely used CAs. The main root certificates for all these are supplied with Conga and can be found in **[DYALOG]/PublicCACerts**. However, most CAs have additional certificates available for download, some of which are application-specific; the latest certificates can be downloaded from the CA's websites.

Table 5-1: Root Certificates for the most widely-used CAs

Authority	Download Root Certificates From...
VeriSign, Geotrust & Thawte	http://www.verisign.com/support/roots.html
Comodo	http://www.comodo.com/repository/
GoDaddy & ValiCert	https://certs.godaddy.com/Repository.go
Cybertrust	http://cybertrust.omniroot.com/support/sureserver/rootcert_ap.cfm
Entrust	http://www.entrust.net/developer/index.cfm

Table 5-1: Root Certificates for the most widely-used CAs (continued)

Authority	Download Root Certificates From...
CAcert	http://www.cacert.org/index.php?id=3
GlobalSign	https://www.globalsign.com/support/root-certificate/osroot.htm
IPS Servidores	http://www.ips.es/Declaraciones/NuevasCAS/NuevasCAS.html The root certificate is not included in [DYALOG]/PublicCACerts .

Conga recognises files with one of the extensions **.cer**, **.pem** or **.der** as certificates. These files must contain data in either PEM or DER format. See *Appendix B* for more information and instructions on how to create certificate files.

5.2 Client and Server Certificates

Client and server certificates are used to verify the identity of the machines at each end of a secure connection (peers). Conga uses X.509 certificates to establish the identity of the peer in a TLS/SSL connection. An X.509 certificate contains information about the certificate subject and the certificate issuer (the CA that issued the certificate), including details of the public key algorithm and the issuer's digital signature.

Dyalog includes a set of test certificates that can be used to test SSL support – these are used by the `TestSecure*` functions in the `Samples` namespace (see *Section 6.1.1*). The test certificates are found in **[DYALOG]/TestCertificates**, which has three subfolders called **ca**, **client** and **server**. These test certificates can be used for testing your own code, but must not be used in production code. The provided certificates are:

- **TestCertificates/ca/ca-key.pem**
The private key for the test CA. Used to sign the client/server and CA certificates. As this is distributed with Conga, no certificate that relies on this can be considered truly secure.
- **TestCertificates/ca/ca-cert.pem**
The public certificate for the test CA. Used to authenticate the client/server certificates.
- **TestCertificates/ca/DyalogCaPublic.pem**
The public certificate for the test CA for <https://ssltest.dyalog.com/>, a Dyalog supplied test site used by the `Samples.TestSecureWebClient` function. It uses a different CA key to the one used by **TestCertificates/ca/ca-cert.pem**.
- **TestCertificates/client/client-cert.pem** and **client-key.pem**
The certificate/key pair used for sample clients.
- **TestCertificates/server/server-cert.pem** and **server-key.pem**
The certificate/key pair used for sample servers.

5.2.1 Certificate Stores



This only applies when running on the Microsoft Windows operating system and is limited to client-side certificates.

Certificates can be stored in common repository known as a *certificate store*. Conga supports the ability to read certificates from the Microsoft certificate store (both client and server must be running on Microsoft Windows).

5.2.2 Revocation Lists

Conga does not support the use of Certificate Revocation Lists. However, this functionality could be added in a future version if required.

5.3 Creating a Secure Client



A client must be defined as secure when it is created – it is not possible to convert an existing non-secure client into a secure client.

Conga creates secure clients by passing certificate and key information to the `DRC.Clt` function – this should be done through the `DRC.X509Cert` class.

EXAMPLE – ASSUMES SECURE SERVER ALREADY ESTABLISHED

```
cert<-DRC.X509Cert.ReadCertFromFile'path/client/client-
cert.pem'
cert.KeyOrigin<-'DER' ('path/client/client-key.pem')
certs<=('X509' cert)('SSLValidation' 16)
DRC.Clt 'C1' 'localhost' 713 'Text', certs
0 C1
```

In this example, the first line of code uses the `ReadCertFromFile` method in the `DRC.X509Cert` class to read the certificate called **client-cert.pem** and record all of its information in a new instance of the `X509Cert` class. The second line specifies the location of the **client-key.pem** file, which contains the private key. The third line creates a variable comprising the parameters required for the creation of the secure client and the fourth line includes this information when creating the secure client.

Alternatively, the locations of the certificate and key files can be explicitly specified:

```
certs<-c('PublicCertFile' ('DER'
('path/client/clientcert.pem'))))
certs,<-c('PrivateKeyFile' ('DER' ('path/client/client-
key.pem'))))
certs,<-c('SSLValidation' 16)
```



```
DRC.Clt 'C1' 'localhost' 713 'Text', certs
0 C1
```

where `PublicCertFile` and `PrivateKeyFile` identify the files containing the public certificate and private key respectively.

The `SSLValidation` parameter that is included when creating a secure client indicates the TLS flags that control the certificate checking process (see *Appendix C* for a complete list of TLS flag values). A typical flag value for a client connection is 16 (accept the server certificate even if its hostname does not match the one it was trying to connect to).

A certificate is not mandatory when creating a secure client; many secure servers accept connections from clients without certificates (known as *anonymous connections*). In this situation, although the server cannot verify the identity of the client, the connection is still encrypted and safe from tampering. Most commercial web sites use anonymous connections as they mean that sensitive data is protected when transmitted over the internet but customers are not required to have a digital signature. To enable an anonymous connection, an empty certificate can be created as follows:

```
'X509' (NEW DRC.X509Cert)
X509 #.[X509Cert]
```



EXAMPLE

```
args←'C1' 'ssltest.dyalog.com' 713 'Text' 100000
args,←('X509' (NEW DRC.X509Cert))
DRC.Clt args
0 C1
```

This is successful even though the `RootCertDir` property has not been explicitly set; in this situation Conga uses Dyalog's certificate in the Microsoft Certificate Store's trusted root certificates directory.



If a secure server's `RootCertDir` parameter has not been defined to point to a valid CA certificate, then a client will be unable to make a secure connection to that server.

EXAMPLE

```
args←'C1' 'ssltest.dyalog.com' 713 'Text' 100000
args,←('X509' (NEW DRC.X509Cert))
DRC.Clt args
1202 ERR_INVALID_PEER_CERTIFICATE /* The peers certificate
is not valid */ 66
```

Without access to a valid CA certificate, validation fails. However, the connection is successful if validation is disabled; this means that, when trying to determine why a connection is failing, it can be useful to set the value of the `SSLValidation` parameter to 32 (accept the server certificate without validating it):

```
args←'C1' 'ssltest.dyalog.com' 713 'Text' 100000
args,←('X509' (NEW DRC.X509Cert))
DRC.SetProp '.' 'RootCertDir' 'path/TestCertificates/ca/'
0
DRC.Clt args,←'SSLValidation' 32
0 C1
```

Having connected without validation, the certificate information can be retrieved and a decision made whether to proceed with the conversation with this server.

EXAMPLE

```
rc cert←DRC.GetProp 'C1' 'PeerCert'
,[1.5]1>cert.Formatted.(ValidFrom ValidTo Issuer Subject)
Wed Apr 01 15:28:02 2015
Fri May 04 17:32:04 2018
C=BE,O=GlobalSign nv-sa,CN=GlobalSign Organization Validation CA -
SHA256 - G2
C=GB,ST=Hampshire,L=Alton,OU=IT,O=Dyalog Limited,CN=*.dyalog.com
```

Once a secure server and client have been linked, operations are exactly the same as for a non-secure server and client.

5.4 Creating a Secure Server



A server must be defined as secure when it is created – it is not possible to convert an existing non-secure server into a secure server.

Secure servers are created in the same way as secure clients (see *Section 5.3*) with the additional rule that a secure server must have a certificate.

EXAMPLE

```
DRC.SetProp '.' 'RootCertDir' 'path\TestCertificates\ca\'
0
cert←DRC.X509Cert.ReadCertFromFile
'path\TestCertificates\server\server-cert.pem'
cert.KeyOrigin←'DER' 'path\TestCertificates\server\server-
key.pem'
certs←('X509' cert)('SSLValidation' 64)
```

```
DRC.Srv 'S1' '' 713 'Text',certs
0 S1
```

When a client is connected to a secure server, the server can request the client's certificate information by calling the `DRC.GetProp` function on the connection object (see *Section A.12*). However, it is not mandatory for a client to have a certificate and a server can only request information about a client's certificate if the `SSLValidation` parameter that is included when creating a secure server indicates (see *Section A.20*) includes one of the following TLS flags (see *Appendix C* for a complete list of TLS flag values):

- *RequestClientCertificate* (64) – including this flag means that connections are permitted from clients even if they do not have a certificate; if a client does have a certificate then information on that certificate is passed to the server.
- *RequireClientCertificate* (128) – including this flag means that connections are only permitted from clients that have a certificate.

If no client certificate is requested, or no certificate exists, then certificate information will have zero rows when queried.

The validation of client certificates requires access to root certificates; before requesting any client certificate information the `DRC.SetProp` function (see *Section A.19*) must be called on the root object to identify the folder containing these certificates. For example:

```
DRC.SetProp '.' 'RootCertDir' 'path\TestCertificates\ca'
```



Connections that are rejected due to certificate validation failure do not generate events on the server, so no application code is required to handle this situation.

5.5 Using the DRC.X509Cert Class

Conga includes a class to encapsulate certificate handling – `DRC.X509Cert`. This class has methods to read certificates from files, folders and Microsoft certificate stores; for a complete description of the `DRC.X509Cert` class, see *Section A.24*.

Certificate information is returned as an `X509Cert` object. This can:

- be used to validate a peer certificate in combination with flags such as `CertAcceptWithoutValidating` (see *Table C-1*).
- enable a server to confirm the identity of a client without requiring a login.

The specific information can vary, but usually includes the certificate issuer, subject, public key algorithm, certificate format version, serial number and valid from/to dates. If no certificate exists or, in the case of a server object, no certificate information has been requested (see *Table C-1*), then the `X509Cert` object is an empty vector.

To read one or more certificates from a file:

```
path←Samples.CertPath
file←path,'client/client-cert.pem'
pmyCert←DRC.X509Cert.ReadCertFromFile file
1
```

As only a single certificate is present, the outermost layer of nesting can be removed:

```
←myCert←myCert
#.DRC.X509Cert.[X509Cert]

myCert.␣NL ␣2           A examine its properties
Cert CertOrigin Elements Extended Formatted KeyOrigin LDRC
ParentCert UseMSStoreAPI
```

`Elements`, `Extended` and `Formatted` contain specific information about the certificate. `Elements` contains the information in a basic format while `Formatted` and `Extended` have the same elements in a more human-readable format (`Extended` may, in some instances, contain more information). For example:

```
myCert.Elements.␣NL ␣2
AlgorithmID AlgorithmParams Description EnhancedKeyUsage
Extensions FriendlyName Issuer IssuerID Key KeyContainer
KeyHex KeyID KeyLength KeyParams KeyProvider KeyProviderType
SerialNo Subject SubjectID ValidFrom ValidTo Version

myCert.Elements.(ValidFrom ValidTo)
2008 2 15 16 19 50 0 2018 2 12 16 20 4 0

myCert.Formatted.(ValidFrom ValidTo)
Fri Feb 15 11:19:50 2008
Mon Feb 12 11:20:04 2018

myCert.Extended.(ValidFrom ValidTo)
Fri Feb 15 16:19:50 2008
Mon Feb 12 16:20:04 2018

myCert.␣NL ␣3           A examine methods
AsArg Chain CopyCertificationChainFromStore IsCert
ReadCertFromFile ReadCertFromFolder ReadCertFromStore Save

mycert.IsCert A my certificate is indeed a certificate!
1
```

5.5.1 Certificate Chains

A *certificate chain* (also known as a *certification path*) is a hierarchy of certificates used for authentication. The first entity in the chain is the certificate for a specified Conga object. Progressing through the chain, each certificate is signed by the owner of the next entity in the chain (that is, the **Issuer** of the lower certificate is the **Subject** for the certificate above it in the chain), as shown in *Figure 5-1*. The final entity in the chain is a root CA certificate.

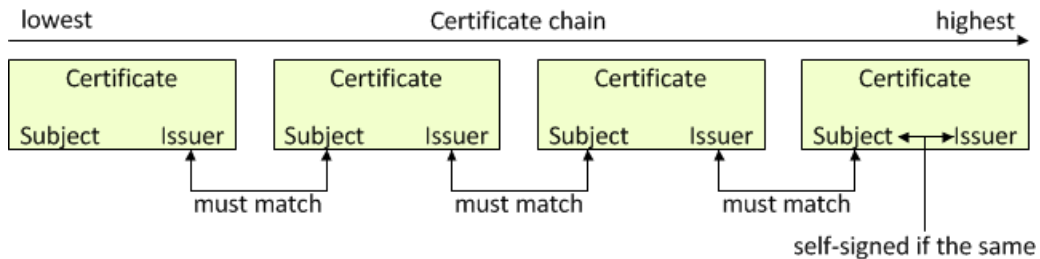


Figure 5-1: Certificate chain

Continuing the example in *Section 5.5*:

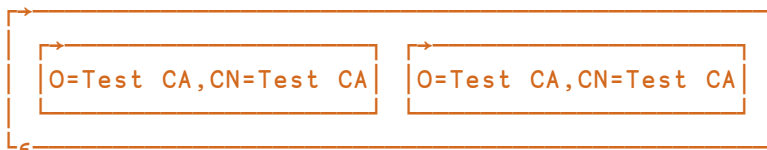
```
myCert.Chain
#.DRC.X509Cert.[X509Cert]  #.DRC.X509Cert.[X509Cert]
```

This is the certificate chain for the test client certificate; it comprises two certificates, the client certificate and the test CA certificate. This can be verified by inspecting the `Issuer` for the lower certificate in the chain and the `Subject` for the higher certificate in the chain – they should be the same:

```
]DISPLAY myCert.Chain.Formatted.Subject
```



```
]DISPLAY myCert.Chain.Formatted.Issuer
```



The lowest certificate in the chain (in this case, the client certificate) is always shown on the left, and the highest certificate in the chain (in this case, the test CA certificate) is always shown on the far right, with any intermediate certificates in the appropriate

location between them. In this example, the `Issuer` for the lower certificate in the chain and the `Subject` for the higher certificate in the chain can be seen to be the same, confirming the authenticity of the client certificate.

In addition, the test CA certificate has the same value for its `Issuer` as it does for its `Subject`; this certificate is, therefore, *self-signed*.

6 The Conga Workspace

The `conga` workspace includes several working examples that demonstrate how to use most of the functionality provided by Conga. Although these are simple, many provide a useful starting point for communicating with Dyalog applications.

Table 6-1 summarises the contents of the `conga` workspace.

Table 6-1: Contents of the *conga* workspace

Name	Description
DRC	Namespace for the Conga interface functions and methods – these are detailed fully in <i>Appendix A</i> .
FTPClient	Class that implements a <i>passive mode</i> FTP client, with functions to: <ul style="list-style-type: none"> • <code>List</code> the contents of a folder on an FTP server • <code>Get</code> and <code>Put</code> files (in either binary mode or text mode) For more information, see <i>Section 6.4</i> .
HTTPUtils	Namespace for manipulating HTTP headers.
RPCServer	Namespace comprising a framework for a Remote Procedure Call server based on <i>Command</i> mode clients for communication between APL systems. For more information, see <i>Section 6.3</i> .
Samples	Namespace for functions that demonstrate and test everything else in the <code>conga</code> workspace. For more information, see <i>Section 6.1</i> .
TODServer	Namespace comprising a simple TOD (Time Of Day) service. For more information, see <i>Section 6.5</i> .
WebServer	Namespace comprising a basic HTTP server that can provide simple Web Services. For more information, see <i>Section 6.2</i> . For <code>MiServer</code> , a more complete APL-based web server, see https://www.github.com/Dyalog/MiServer .

6.1 Namespace: Samples

The `Samples` namespace contains functions and operators that interact with web servers and services as well as functions that demonstrate and test the majority of the functionality provided in the `conga` workspace. Some of the functions within the `Samples` namespace are not documented; these are called by other functions and should not be amended.

6.1.1 Function: Samples.Test*

The `Samples.Test*` functions illustrate some of the functionality and versatility of the contents of the `conga` workspace. They comprise:

- **TestAll**
Cover function that runs several of the other `Test*` functions. The final two tests require a TOD (Time of Day) and QOTD (Quote of the Day) server to be running respectively.
- **TestAllSecure**
Secure version of the `TestAll` function.
- **TestCompression**
Uses the `DRC.Flate.Inflate` and `DRC.Flate.Deflate` functions (see *Section A.10* and *Section A.9* respectively) to apply and remove HTTP deflate compression algorithms.
- **TestFTPClient**
Uses the `FTPClient` class to connect to <ftp.mirror-service.org> and accesses the file `pub/FreeBSD/README.TXT` from this website.
- **TestRPCServer**
Starts an RPC server on port 5050 and then spawns a number of threads; each thread makes a remote procedure call to one of the functions (`Foo` and `Goo`) in the `RPCServer` namespace (see *Section 4.5.1* and *Section 6.3*).
- **TestSecureConnection**
Creates a secure server and connects a secure client to it; sends a transaction over the secure connection.
- **TestSecureWebClient**
Secure version of the `TestWebClient` function.
- **TestSimpleServices**
Attempts to connect to and use the TOD (Time of Day) and QOTD (Quote of the Day) services on a named host.
- **TestWebClient**
Uses the `Samples.HTTPGet` function (see *Section A.26*) to retrieve the contents of Dyalog Ltd's website (<http://www.dyalog.com/>).
- **TestWebFunctionServer**
Launches the example web server, starts a number of threads and calls the

`Samples.HTTPGet` function to request pages, thus testing that the web server that has been started and is responding as expected.

- `TestX509Certs`
Builds certificates and key files into variables used by the secure versions of the tests.

6.2 Namespace: WebServer

This namespace contains a basic implementation of a web server. Although this web server is too simple to provide many of the services that more sophisticated web servers provide, it does illustrate how a web server that interfaces to a web browser can be implemented with only a very small amount of APL; even this very simple web server can deliver real files from the file system or, using APL functions, intercept requests and manufacture virtual pages on request.



For a more complete APL based web server, see Dyalog's MiServer at <https://www.github.com/Dyalog/MiServer>.

The `WebServer.Run` function launches the web server (see *Section A.30*).

6.2.1 Function: WebServer.Run

The code for the `WebServer.Run` function works as follows:

- [10] Call the `DRC.Init` function to ensure that the DRC namespace is initialised.
- [28] Create a server object on the specified port in *Raw* mode (this is the same as *Text* mode except it returns byte numbers in the range 0-255).
- [39] Loop on the `DRC.Wait` function, timing out every 10 seconds – this allows for graceful shutdowns as well as housekeeping.
- [41] Check the first element of the result of the `DRC.Wait` function; this is the return code.
- [42] If the return code is 0, then the `DRC.Wait` function successfully returned an event.
- [45–50] If the event was an error on the socket, then that socket must be closed and the data namespace for the client cleaned up (the `SpaceName` function generates a name for the namespace based on the IP address and port number of the client).

- [52–55] If the event was the receipt of data, then call the `WebServer.HandleRequest` function in a new thread (in the appropriate client namespace), passing it the object name and input data. The `WebServer.HandleRequest` function calls the `DRC.Send` function to send the response to the client.
- [57–59] If the event was the client closing the connection, then expunge the namespace.
- [61–71] If the event was a `Connect` event, then create a unique namespace for the connection, assigning local variables for server and client (including certificates if the connection is a secure connection).
- [73–74] If the return code is 100, then nothing has happened for 10 seconds (the timeout period set in [39]). Optionally, housekeeping tasks can be inserted here. For a busy web server, housekeeping is necessary even without timeouts.
- [76–78] If the return code is 1010 (object not found), then the server object cannot be located. The most likely explanation is that another thread has closed it – several of the `Samples.Test*` functions (see *Section 6.1.1*) do this once they have completed client tests.
- [88–89] The loop is exited when a component of the server initiates a shut down. The `DRC.Close` function is called to close the server object.

6.3 Namespace: RPCServer

The RPC server is similar to the web server discussed in *Section 6.2*, except that *Command* mode is used to transmit RPCs to the server; the server then validates and executes them and returns the array result to the client.

As *Command* mode is used, both the client and the server need to be Conga users; constructing a non-APL client that can use *Command* mode is possible but not trivial. However, most non-APL clients already support SOAP for remote procedure calls (SOAP is an established standard for web services for which there are many tools in the non-APL world) and SAWS, Dyalog's Stand-Alone Web Services framework, enables users to provide and consume SOAP-based web services. For more information, see the *SAWS User Guide*.

6.3.1 Function: RPCServer.Run

The code for the `RPCServer.Run` function works as follows:

- [10] Call the `DRC.Init` function to ensure that the DRC namespace is initialised.
- [24–31] So that it can return an error if it is unable to start the server, the `DRC.Srv` function first creates a *Command* mode server on line [24]; the `Run` function only starts a new handling thread on line [26] (by calling itself recursively with a left argument of 0) if the server was successfully created. The handler continues execution from the `:While` on line [32].
- [32–33] Loop on the `DRC.Wait` function, timing out every 3 seconds – this allows for graceful shutdowns as well as housekeeping.
- [35] Check the first element of the result of the `DRC.Wait` function; this is the return code.
- [36] If the return code is 0, then the `DRC.Wait` function successfully returned an event.
- [38–42] If the event was an error, then:
 - if the object in error was the server, close it and stop running.
 - if the object in error was not the server, ignore the error
 (Housekeeping tasks could be performed here if client sessions were being tracked.)
- [44–53] If the event was the receipt of data, then validate the format of the incoming array to confirm that the first element names a function that can be called. If all is OK, run the `Process` function in a new thread, passing the object name and input data to it. The `Process` function calls the `DRC.Progress` function to signal progress and then the `DRC.Respond` function to send the answer to the client.
- [55–62] If the event was a `Connect` event then:
 - for non-secure connections – ignore.
 - for secure connections – retrieve and display peer certificate information (serial number, issuer and subject).
- [67] If the return code is 100, then nothing has happened for 3 seconds (the timeout period set in [16]). Optionally, housekeeping tasks can be inserted here. For a busy web server, housekeeping is necessary even without timeouts.

- [69–70] If the return code is 1010 (object not found), then the server object cannot be located. The most likely explanation is that another thread has closed it – several of the `Samples.Test*` functions (see *Section 6.1.1*) do this once they have completed client tests.
- [76–78] The loop is exited when a component of the server initiates a shutdown. The `DRC.Close` function is called to close the server object.

6.4 Class: FTPClient

This class implements a basic passive mode FTP client. The `Samples.TestFTPClient` function shows an example of the `FTPClient` class in use – it lists the contents of the **pub/FreeBSD** folder at ftp.mirrorservice.org and retrieves the **README.TXT** file from this folder.

This code for the `FTPClient` class works as follows:

Open [4]	Call the <code>DRC.Init</code> function to ensure that the DRC namespace is initialised.
Open [6]	Create a client object in <i>Text</i> mode for issuing commands to the FTP server.
Open [11–13]	Use the <code>Do</code> method to enter user ID and password and check for the expected responses from the server.
Do [5&10]	(called by the <code>Open</code> method) Send a command to the server and return the FTP state code following the command using the <code>ReadReply</code> method.
ReadReply [2]	(called by the <code>Do</code> method) Wait for a response from the server.
GetData [3]	Execute the <i>PASV</i> method to prepare for passive-mode data transfer; the server returns the dataport that it has opened.
GetData [4]	Create a client object in <i>Text</i> or <i>Raw</i> mode to the dataport identified by the <i>PASV</i> method.
GetData [5]	Specify whether the data will be transferred using ASCII or binary format.
GetData [6]	Issue the command to the server to start sending data.

GetData [9-11]	Continue collecting output response until the server closes the connection.
GetData [14]	Confirm that the server thinks transfer of data has been completed.
PutData [6-8]	Same as GetData[3-5].
PutData [10]	Send all data in a single call to the DRC . Send function.

6.5 Namespace: TODServer

This namespace comprises a simple TOD (Time Of Day) service in the form of a server object that is created and closed by a Run function.



This TOD server is the starting point for the examples in *Section 4.3* and *Section 4.4*.

6.5.1 Function: TODServer.Run

The code for the TODServer .Run function is:

```

▽ Run port;wait;data;event;obj;rc;r
[1]  A Time of Day Server Example (use port 13 by default)
[2]
[3]  ##.DRC.Init ' ' ♦ DONE←0 A DONE is used to stop service
[4]  :If 0≠1>r←##.DRC.Srv 'TOD' ' ' port 'Text'
[5]  []←'Unable to start TOD server: ',⌞r
[6]  :Else
[7]  []←'TOD Server started on port ',⌞port
[8]  :While ~DONE
[9]  rc obj event data←4↑wait←##.DRC.Wait 'TOD' 1000 A
Time out every second
[10] :Select rc
[11] :Case 0
[12] :Select event
[13] :Case 'Connect'
[14] r←(,'ZI2,<:>,ZI2,<:>,ZI2,< >,ZI2,<->,ZI2,<->,ZI4'
[]FMT 1 6p[]TS[4 5 6 3 2 1]),[]AV[4 3]
[15] {}##.DRC.Send obj r 1 A 1=Close connection
[16] :Else
[17] {}##.DRC.Close obj A Anything unexpected

```

```

[18]             :EndSelect
[19]             :Case 100  A Time out - Housekeeping Here
[20]             :Else
[21]                 []←'Error in Wait: ',⌘wait ♦ DONE←1
[22]             :EndSelect
[23]         :EndWhile
[24]         {}##.DRC.Close'TOD' ♦ []←'TOD Server terminated.'
[25]     :EndIf
▽

```

This works as follows:

- [3] Call the `DRC.Init` function and set global flag `DONE` to zero.
- [4] Create a server object named `TOD` on selected port in *Text* mode.
- [8] Repeat the following until `DONE` is equal to 1:
- [9] Wait for any event and split the result into `rc` (return code), `obj` (object name – a string identifying a child object of `TOD` with a name like 'TOD.CON00000000'), `event` and `data`.
- [14] If return code was 0 and the event was `Connect`, then format the time of day.
- [15] Send the time of day to `obj`. The third element of the argument is set to a value of 1 to instruct Conga to close the object as soon as the data has been sent.
- [17] For any other event, close the connection.
- [19] Check here every 1,000 milliseconds (as specified in the argument to the `DRC.Wait` function on line [9]) for housekeeping tasks to perform (none are specified in this code sample, but they can be added here if required).
- [21] Any return code from the `DRC.Wait` function other than 0 or 100 shuts down the service.
- [24] Close the server object.

A Technical Reference

The functions and methods in the DRC namespace are intended for use by applications; these are documented in this appendix. Any additional functions in the DRC namespace are for internal use and should not be called by application code.

The notation used when describing the syntax for a function/method in this document is as follows:

- square brackets [] indicate an optional argument
- curly braces { } indicate a mandatory argument
- a vertical line | separates mutually exclusive arguments
- italic text indicates an element that must be populated by the user
i>

The order in which parameters are specified must be as shown in the syntax; however, individual parameters can be specified using parenthesised name-value pairs to eliminate the need to specify all parameters, for example, ('X509' myCert) or ('EOM' (␣UCS 13 10)).

A.1 DRC Return Codes

Many of the functions in the DRC namespace generate a *return code* as the first element of their result. The value of this return code indicates whether the function was successful in its action:

- If the return code is 0, then the function successfully performed its requisite actions; the rest of the result is as described in this appendix.
- If the return code is not 0, then the function did not successfully perform its requisite actions; the rest of the result vector comprises *error name* (vector element [2]) and *error description*, if available (vector element [3]).

Dyalog Ltd recommends that you check the return code in the result before attempting to process other elements of the result. As the number of items returned can vary depending on whether the function successfully performed its requisite actions, this requires code resembling the following:

```
:if 0≠1↑res ← DRC.Certs arg
    rc err desc ← res
    ...                               A error processing
:else
    rc stores ← res
    ...                               A normal processing
```

A.2 Function: DRC.Certs



This only applies when running on the Microsoft Windows operating system and is limited to client-side certificates.

Purpose: Returns a list of all the certificate store names defined on the local machine.

Syntax: `rc stores ← DRC.Certs {'ListMSStore'}`

where:

- `rc` is the return code (see *Section A.1*)
- `stores` is a vector of character vectors each containing a store name, for example, **My** and **Root**.
- `'ListMSStore'` is an instruction to the function to include Microsoft certificate store names in the returned `stores` vector

Example:



A.3 Function: DRC.ClientAuth



Integrated Windows Authentication is only available on a Microsoft Windows domain – both client and server must be running on Microsoft Windows.

Purpose: Performs client-side Integrated Windows Authentication (IWA). Only valid for a *Command-mode* client connected to a *Command-mode* server

Syntax: `rc ← DRC.ClientAuth {clientname} [{userid} {password}]`

where:

- `rc` is the return code (see *Section A.1*)
- `clientname` is the name of the *Command-mode* client.
- `userid` is the user's Microsoft Windows User ID.
- `password` is the user's Microsoft Windows password.



DRC.ClientAuth and DRC.ServerAuth (see *Section A.18*) must be run at the same time.

A.4 Function: DRC.Close

Purpose: Closes the specified Conga object.

Syntax: `rc ← DRC.Close
{servername|clientname|connectionname|commandname}`

where:

- `rc` is the return code (see *Section A.1*)
- `servername|clientname|connectionname|commandname` is the name of the Conga server/client/connection/command to close (respectively).

Example:

```
DRC.Close 'C1'
0
```

A.5 Function: DRC.Clt

Purpose: Creates a Conga client object.

Syntax: `rc name ← DRC.Clt {clientname} {Address} [Port] [Mode]
[BufferSize] [SSLValidation] [EOM] [IgnoreCase] [Protocol]
[PublicKeyData] [PrivateKeyFile] [PrivateKeyPass]
[PublicKeyFile] [PublicKeyPass] [PrivateKeyData] [Priority]
[Magic] [X509]`

where:

- `rc` is the return code (see *Section A.1*)
- `name` is the name of the Conga client that has been created. If no `clientname` was specified (`' '`) then this is auto-generated.
- `clientame` is the name of the Conga client to create. If `' '` is specified rather than a specific name, then the name will be auto-generated.
- `Address` is the address of the server
- `Port` is the port number that the server will listen on
- `Mode` is the connection protocol (see *Section 4.1.3*). Possible values are *Command*, *Text*, *Raw*, *BlkText* or *BlkRaw*. The default is *Command*.
- `BufferSize` is the maximum size (in bytes) allocated to the buffer that receives data transmissions. The default is 16,384. Only valid for clients in *Raw/Text/BlkRaw/BlkText* mode, not those in *Command* mode.
- `SSLValidation` is the sum of the relevant TLS flags (see *Appendix C*). Only valid when creating a secure client (see *Section 5.3*).
- `EOM` is a simple character vector or a vector of vectors indicating the termination string(s) (see *Section 4.1.3*). Only valid for clients in *Raw/Text* mode, not those in *BlkRaw/BlkText/Command* mode.
- `IgnoreCase` is a Boolean indicating whether searches for the termination string defined in `EOM` are case sensitive. Possible values are:
 - `0` : do not ignore case when searching for termination strings
 - `1` : ignore case when searching for termination strings

Only valid for clients in *Raw/Text* mode, not those in *BlkRaw/BlkText/Command* mode.

- `Protocol` is the communication protocol to use. Possible values are:
 - `IPv4` : use the IPv4 connection protocol; if this is not possible then generate an error
 - `IPv6` : use the IPv6 connection protocol; if this is not possible then generate an error
 - `IP` : use the IPv6 connection protocol; if this is not possible then use the IPv4 connection protocol.

The default is to inherit the protocol defined for the root object.

- `PublicCertData` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).
- `PrivateKeyFile` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).

- `PrivateKeyPass` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).
- `PublicCertFile` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).
- `PublicCertPass` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).
- `PrivateKeyData` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).
- `Priority` is the GnuTLS priority string (for complete documentation of this, see <http://www.gnutls.org/manual/gnutls.html#Priority-Strings>).
- `Magic` is a 32-bit integer used to check that the data has not been corrupted. Only valid for clients in *BlkRaw/BlkText* mode, not those in *Raw/Text/Command* mode.
- `X509` is a reference to an instance of the `X509Cert` class. Only valid when creating a secure client (see *Section 5.3*).

The `[PublicCertData]`, `[PrivateKeyFile]`, `[PrivateKeyPass]`, `[PublicCertFile]`, `[PublicCertPass]` and `[PrivateKeyData]` arguments are mutually exclusive with the `[X509]` argument.

Examples:

To create `C1`, a *Command* mode client of a server at 192.168.1.1 listening on port 5050:

```
DRC.Clt 'C1' '192.168.1.1' 5050
0 C1
```

To create a secure *Command* mode client of the secure *Command* mode server at 192.168.1.1 listening on port 5050:

```
mycert<=>DRC.X509Cert.ReadCertFromFile 'path/client-cert.pem'
mycert.KeyOrigin<'DER' 'path/client-key.pem'
DRC.Clt 'C1' '192.168.1.1' 5050,=('X509' mycert)
('SSLValidation' 16)
0 C1
```

To create a *Text* mode client (with an auto-generated name) of a server on the same machine, listening on port 30, with a maximum buffer size of 1000 characters and a termination sequence of `<CRLF>`:

```
DRC.Clt '' 'localhost' 30 'Text' 1000 ('EOM' (␣UCS 13 10))
0 CLT00000000
```

Related function:

- `DRC.Srv` – see *Section A.20*

A.6 Function: DRC.Describe

Purpose: Returns a description of the specified Conga object. Similar to the `DRC.Tree` function (see *Section A.21*) except that `DRC.Describe` only returns information for the specified object (not its children) and the descriptions are textual rather than numeric codes.

Syntax: `rc desc ← DRC.Describe {objectname}`

where:

- `rc` is the return code (see *Section A.1*)
- `desc` is a multi-element vector providing a textual description of the specified object
- `objectname` is the name of the Conga object to describe

For all objects except the root object, the description `desc` starts with the following elements:

- `[1]` is the name of the object
- `[2]` is the object's type (see *Section 4.1.1*)
- `[3]` is the object's state (see *Section 4.1.2*)

The description `desc` of objects of type 4 (commands) and 5 (messages) has additional elements:

- `[4]` is the size of the object that has been processed so far (in bytes)
- `[5]` is the size of the object that has not yet been processed (in bytes)

For the root object `'.'`, the description `desc` comprises the following elements:

- `[1]` is [DRC]
- `[2]` is the version of Conga
- `[3]` is the object's state (see *Section 4.1.2*)
- `[4]` is the thread count

Examples:

```
DRC.Describe '.'
0 [DRC] Conga Dynamic Link Library 2.6.956.0 Copyright (C)
2004-2015 Dyalog Ltd. built Nov 2 2015 10:48:51. GnuTLS 3.2.15
Copyright (c) 2000-2015 Free Software Foundation, Inc. Built Jul
9 2015 at 09:48:37. Revision: 106 State=RootInit Threads=0
```

```
DRC.Describe 'C1'
0 CLT00000000 Client Connected
```

Similar functions:

- `DRC.Names` – see *Section A.14*
- `DRC.Tree` – see *Section A.21*

A.7 Function: DRC.Error

Purpose: Converts an error number into a textual identification or description of the error.

Syntax: `no name [desc] ← DRC.Error {no}`

where:

- `no` is the error number
- `name` is the name of the error
- `desc` is a description of the error, for example, `/* unable to complete a TLS handshake with the peer */`

Example:

```
DRC.Error 1009
1009 ERR_NAME_IN_USE
```

A.8 Function: DRC.Exists

Purpose: Verifies whether a specified Conga object exists.

Syntax: `bool ← DRC.Exists {objectname}`

where:

- `bool` indicates whether the object exists. Possible values are:
 - 0 – the object does not exist
 - 1 – the object exists
- `objectname` is the name of the Conga object whose existence is being verified

Example:

```
DRC.Exists 'C1'
1
```

A.9 Method: DRC.Flate.Deflate

Purpose: Compresses data using the deflate compression scheme (see *Section 4.6*). Used to implement server-side HTTP compression.

Syntax: `comp ← DRC.flate.Deflate {data}`

where:

- `comp` is the compressed data. Comprises an integer vector with values in the range 0-255. The first 2 elements comprise a header, 120 156, for the zlib wrapper for the compressed data – this header should be removed before sending to the client.
- `data` is the data to be compressed. Comprises an integer vector with values in the range 0-255. To convert from character format to integer format, use 'UTF-8' → UCS `data`.

Example:

```
DRC.flate.Deflate 256|83 →DR 2000p'this is a test'~
120 156 43 201 200 44 86 0 162 68 133 146 212 226 146 146 81 222
40 111 148 55 202 27 229 141 242 70 121 67 144 7 0 17 217 213 243
```

A.10 Method: DRC.Flate.Inflate

Purpose: Decompresses data that has been compressed using the deflate compression scheme (see *Section 4.6*). Used to implement client-side HTTP compression.

Syntax: `data ← DRC.flate.Inflate {comp}`

where:

- `data` is the decompressed data. Comprises an integer vector with values in the range 0-255. To convert from integer format to character format, use 'UTF-8' → UCS `data`.
- `comp` is the compressed data. Comprises an integer vector with values in the range 0-255. The first 2 elements comprise a header, 120 156, for the zlib wrapper for the compressed data – if the header is not present, it should be prepended before the `DRC.flate.Deflate` method is called.

Example:

```
tmp ← DRC.flate.Deflate 256|83 →DR 'this is a test'
'UTF-8' →UCS DRC.flate.Inflate 256|83 →DR tmp
this is a test
```

A.11 Method: DRC.Flate.IsAvailable

Purpose: Validates whether the deflate compression library is loaded.

Syntax: `bool ← DRC.flate.IsAvailable`

where:

- `bool` is a Boolean indicating whether the deflate compression library is loaded:
 - 0 : the deflate compression library is not loaded and compression cannot be used
 - 1 : the deflate compression library is loaded

Example:

```
DRC.flate.IsAvailable
1
```

A.12 Function: DRC.GetProp

Purpose: Retrieves property values for the specified Conga object.

Syntax: `rc res ← DRC.GetProp {objectname} {property} [arg]`

where:

- `rc` is the return code (see *Section A.1*)
- `res` is the retrieved property value; its format depends on the property requested (see *Table A-1*)
- `objectname` is the name of the Conga object for which to retrieve the value of the specified property
- `property` is the name of the property to retrieve the value for; not all properties are relevant for all object types (see *Table A-1*)
- `arg` is an additional argument that might be required depending on the property requested (see *Table A-1*)

The property values that can be retrieved depend on the type of the specified Conga object; they are described in *Table A-1*. These values are specified using the `DRC.SetProp` function (see *Section A.19*) or when a server/client is created using the `DRC.Srv/DRC.Clt` function (see *Section A.20* and *Section A.5* respectively).

Table A-1: Properties that can be retrieved by the *DRC.GetProp* function

Property	Object Type	Description/Syntax
KeepAlive	Client, Server, Connection	The frequency with which periodic heartbeat messages are sent to verify that the connection is live. A 2-element vector in which: <ul style="list-style-type: none"> [1] is the time (in ms) to wait before sending the first heartbeat [2] is the time interval (in ms) between heartbeats
LocalAddr	Client, Server, Connection	A 4-element vector in which: <ul style="list-style-type: none"> [1] is the communication protocol [2] is the IP address (formatted according to the communication protocol) and port number [3] is the address bytes [4] is the port number being used
Magic	Connection	A 32-bit number unique to the blocks in a single data transmission. Only valid in <i>BlkRaw/BlkText</i> mode, not <i>Raw/Text/Command</i> mode.
OwnCert	Client, Server, Connection	The X509Cert object containing information about the certificate of the specified Conga object.
PeerAddr	Client, Connection	The address of the specified Conga object's peer. A 4-element vector in which: <ul style="list-style-type: none"> [1] is the communication protocol [2] is the IP address (formatted according to the communication protocol) and port number [3] is address bytes [4] is the port number being used
PeerCert	Client, Connection	The X509Cert object containing information about the certificate of the specified Conga object's peer.
PropList	all	A list of the properties relevant for the object's type.

Table A-1: Properties that can be retrieved by the *DRC.GetProp* function (continued)

Property	Object Type	Description/Syntax
Protocol	Root	<p>The communication protocol in use. Possible values are:</p> <ul style="list-style-type: none"> IPv4 : use the IPv4 connection protocol; if this is not possible then generate an error IPv6 : use the IPv6 connection protocol; if this is not possible then generate an error IP : use the IPv6 connection protocol; if this is not possible then use the IPv4 connection protocol <p>This property value is inherited when creating a connection unless a different value is specified.</p>
ReadyStrategy	Root, Server	<p>The strategy by which the next connection to process is determined when more than one connection has received data. Possible values are:</p> <ul style="list-style-type: none"> 0 : "Use First" – process the first connection in the object tree (for information on the object tree, see <i>Section A.21</i>). This approach can result in connections not being serviced. 1 : "Round Robin" – process the first connection in the object tree after the one that has just been processed. 2: "Oldest First" – process the connection that has been waiting for the longest time. This is considered to be the least biased approach but consumes slightly more CPU than strategy 1.
RootCertDir	Root	Full path to (and name of) the directory that contains Certificate Authority root certificates.

Table A-1: Properties that can be retrieved by the `DRC.GetProp` function (continued)

Property	Object Type	Description/Syntax
TCPLookup	Root	<p>Requires an additional argument comprising a 2-element vector in which:</p> <ul style="list-style-type: none"> [1] is a URL or IP address as a character string [2] is the port number (0 means all ports at the URL/IP address) or service name <p>The address of the specified URL/IP address and port. A 4-element vector in which:</p> <ul style="list-style-type: none"> [1] is the communication protocol [2] is the IP address (formatted according to the communication protocol) and port number [3] is the address bytes [4] is the port number <p>Multiple 4-element vectors can be returned if both IPv4 and IPv6 information is available.</p>

Examples:

```

DRC.GetProp '.' 'PropList'
0  Certificates DecodeCert PropList Protocol ReadyStrategy
RootCertDir Stores TCPLookup

DRC.GetProp '.' 'TCPLookup' 'www.dyalog.com' 80
0  IPv6 [2a02:2658:1012::35]:80 42 2 38 88 16 18 0 0 0 0 0 0
0 0 53 80 IPv4 81.94.205.35:80 81 94 205 35 80

DRC.GetProp 'C1' 'OwnCert'
0  #.DRC.[X509Cert]
```

A.13 Function: DRC.Init

Purpose: Loads and initialises (or reinitialises) the Conga Dynamic Link Library (Microsoft Windows) or Shared Library (UNIX/Linux).

Syntax: `rc ← [reset] DRC.Init {''}`

where:

- `rc` is the return code (see *Section A.1*)
- `reset` is a code indicating the action to take if Conga has already been initialised.

Possible values are:

- 1 : close any existing Conga objects
- -1 : reload the library

For any other value, a message is returned stating that Conga has already been loaded.

The right argument to this function is currently unused but is reserved for future extensions.

Example:

```
DRC.Init ''
0 Conga loaded from: ...\\conga27x64Uni
```

A.14 Function: DRC.Names

Purpose: Returns the names of existing Conga objects that are first-level descendants of the specified Conga object.

Syntax: `names ← DRC.Names {objectname}`

where:

- `names` is a list of the names of all first-level descendants of the specified Conga object. If there are no first-level descendants of the specified Conga object, then `names` is an empty vector.
- `objectname` is a character vector of the name of the Conga object for which to return the names of its first-level descendants

Examples:

```
DRC.Names ''
C1 C2 C3

DRC.(Close''Names '')
0 0 0
```

Similar functions:

- `DRC.Describe` – see *Section A.6*
- `DRC.Tree` – see *Section A.21*

A.15 Function: DRC.Progress

Purpose: Sends an APL array to a client in response to a command received from that client. Only valid for a *Command*-mode server.

A server can call the `DRC.Progress` function any number of times before calling the `DRC.Respond` function (see *Section A.16*).

Syntax: `rc ← DRC.Progress {commandname} {data}`

where:

- `rc` is the return code (see *Section A.1*)
- `commandname` is the name of the command received from the client. It must match the `objectname` returned by the `DRC.Wait` function (see *Section A.23*)
- `data` is any array

Example:

A *Command*-mode client sends data to a *Command*-mode server using the `DRC.Wait` function. The server stores the result in `waitresult`. The following expression can be used to send a progress report to the client:

```
DRC.Progress (2>waitresult) 'Task 50% completed'
```

Related functions:

- `DRC.Respond` – see *Section A.16*
- `DRC.Send` – see *Section A.17*
- `DRC.Wait` – see *Section A.23*

A.16 Function: DRC.Respond

Purpose: Sends an APL array to a client in response to a command received from that client. Only valid for a *Command*-mode server.

Syntax: `rc ← DRC.Respond {commandname} {data}`

where:

- `rc` is the return code (see *Section A.1*)
- `commandname` is the name of the command received from the client. It must match the `objectname` returned by the `DRC.Wait` function (see *Section A.23*)
- `data` is any array

Example:

A *Command*-mode client sends data to a *Command*-mode server using the `DRC.Wait` function. The server stores the result in `waitresult`. The following expression can be used to call the `Process` function on the data that accompanied the most recent command and send the result to the client:

```
DRC.Respond (2>waitresult) (Process 4>waitresult)
```

Related functions:

- `DRC.Progress` – see *Section A.15*
- `DRC.Send` – see *Section A.17*
- `DRC.Wait` – see *Section A.23*

A.17 Function: DRC.Send

Purpose: Send data to the peer client/server. Not valid for a *Command*-mode server.

Syntax: `rc clientname|connectionname.commandname|messagename ← DRC.Send {clientname[.commandname|.messagename]|connectionname} {data} [close]`

where:

- `rc` is the return code (see *Section A.1*)
- `clientname[.commandname]|connectionname` is dependent on the mode whether a full name is supplied or auto-generation is required:
 - for client objects and server objects in *Text* mode or *Raw* mode:
 - if `clientname` is supplied, Conga auto-generates a `messagename` and returns it as the second element of the result in the format `clientname.messagename`.
 - if `connectionname` is supplied, Conga auto-generates a `messagename` and returns it as the second element of the result in the format `connectionname.messagename`.
 - if `clientname.messagename` or `connectionname.messagename` is supplied, Conga returns it unaltered as the second element of the result.



`messagename` is not the name of a message object.

- for client objects in *Command* mode:
 - if `clientname` is supplied, Conga auto-generates a `commandname` and returns it as the second element of the result in the format `clientname.commandname`.
 - if `clientname.commandname` is supplied, Conga returns it unaltered as the second element of the result.
- `data` is the array to send to the peer server/client object.
- `close` indicates the action to take after sending the data to the peer object.
Possible values are:
 - 0 : no action taken. This is the default value.
 - 1 : the connection will be closed.
 - 2 : the connection remains open, the command object will be closed – only relevant for client objects in *Command* mode

Examples:

A client object in Command mode:

To create a command with an auto-generated name below client C1 and send an APL array to the server:

```
DRC.Send 'C1' ('PlusReduce' (10))
0 C1.Auto00000000
```

A server object in Command mode:

Not applicable – server objects in *Command* mode use the `DRC.Respond` function rather than the `DRC.Send` function (see *Section A.16*).

A server or client object in Raw mode or Text mode:

To send the text 'Bye' on client C1 and subsequently close the connection:

```
DRC.Send 'C1' ('Bye', [UCS 13]) 1
0 C1.Auto00000001
```

Related functions:

- `DRC.Progress` – see *Section A.15*
- `DRC.Respond` – see *Section A.16*
- `DRC.Wait` – see *Section A.23*

A.18 Function: DRC.ServerAuth



Integrated Windows Authentication is only available on a Microsoft Windows domain – both client and server must be running on Microsoft Windows.

Purpose: Performs server-side Integrated Windows Authentication (IWA).

Syntax: `rc ← DRC.ServerAuth {connectionname}`

where:

- `rc` is the return code (see *Section A.1*)
- `connectionname` is the name of the connection through which a *Command-mode* server responds to a *Command-mode* client.



`DRC.ClientAuth` (see *Section A.3*) and `DRC.ServerAuth` must be run at the same time.

A.19 Function: DRC.SetProp

Purpose: Updates the qualified properties of the specified Conga object.

Syntax: `DRC.SetProp {objectname} {property} {value}`

where:

- `objectname` is the Conga object for which to set a new property value
- `property` is the name of the property to set
- `value` is the value to set the specified property to

The property values that can be set depend on the type of the specified Conga object; they are described in *Table A-2*. Some of these values can also be specified when a server/client is created using the `DRC.Srv/DRC.Clt` function (see *Section A.20* and *Section A.5* respectively). The current values can be retrieved using the `DRC.GetProp` function (see *Section A.12*).

Table A-2: Properties that can be set by the DRC. *SetProp* function

Property	Object Type	Description/Syntax
KeepAlive	Client, Server, Connection	<p>The frequency with which periodic heartbeat messages are sent to verify that the connection is live. A 2-element vector in which:</p> <ul style="list-style-type: none"> [1] is the time (in ms) to wait before sending the first heartbeat [2] is the time interval (in ms) between heartbeats
Protocol	Root	<p>The communication protocol to use. Possible values are:</p> <ul style="list-style-type: none"> IPv4 : use the IPv4 connection protocol; if this is not possible then generate an error IPv6 : use the IPv6 connection protocol; if this is not possible then generate an error IP : use the IPv6 connection protocol; if this is not possible then use the IPv4 connection protocol <p>The default is IP. This property value is inherited when creating a connection unless a different value is specified.</p>
ReadyStrategy	Root, Server	<p>The strategy by which the next connection to process is determined when more than one connection has received data. Possible values are:</p> <ul style="list-style-type: none"> 0 : "Use First" – process the first connection in the object tree (for information on the object tree, see <i>Section A.21</i>). This approach can result in connections not being serviced. 1 : "Round Robin" – process the first connection in the object tree after the one that has just been processed. 2 : "Oldest First" – process the connection that has been waiting for the longest time. This is considered to be the least biased approach but consumes slightly more CPU than strategy 1. <p>The default is 2.</p>
RootCertDir	Root	Full path to (and name of) the directory that contains Certificate Authority root certificates.

Examples:

```
DRC.SetProp '.' 'RootCertDir' 'C:\..\TestCertificates\ca'
0
```

```
DRC.SetProp 'C1' 'KeepAlive' (1000 2000)
0
```

A.20 Function: DRC.Srv

Purpose: Creates a Conga server to listen on a specified port. If certificate information is provided, then a secure server is created.

Syntax: `rc name ← DRC.Srv {Name} [Address] [Port] [Mode] [BufferSize] [SSLValidation] [EOM] [IgnoreCase] [Protocol] [PublicCertData] [PrivateKeyFile] [PrivateKeyPass] [PublicCertFile] [PublicCertPass] [PrivateKeyData] [Priority] [Magic] [X509]`

where:

- `rc` is the return code (see *Section A.1*)
- `name` is the name of the Conga server that has been created. If no `Name` was specified (' ') then this is auto-generated.
- `Name` is the name of the Conga server to create. If ' ' is specified rather than a specific name, then the name will be auto-generated.
- `Address` is the address of the server
- `Port` is the port number that the server will listen on
- `Mode` is the connection protocol (see *Section 4.1.3*). Possible values are *Command*, *Text*, *Raw*, *BlkText* or *BlkRaw*. The default is *Command*.
- `BufferSize` is the maximum size (in bytes) allocated to the buffer that receives data transmissions. The default is 16,384. Only valid for clients in *Raw/Text/BlkRaw/BlkText* mode, not those in *Command* mode.
- `SSLValidation` is the sum of the relevant TLS flags (see *Appendix C*). Only valid when creating a secure client (see *Section 5.3*).
- `EOM` is a simple character vector or a vector of vectors indicating the termination string(s) (see *Section 4.1.3*). Only valid for clients in *Raw/Text* mode, not those in *BlkRaw/BlkText/Command* mode.

- **IgnoreCase** is a Boolean indicating whether searches for the termination string defined in EOM are case sensitive. Possible values are:
 - 0 : do not ignore case when searching for termination strings
 - 1 : ignore case when searching for termination strings
 Only valid for clients in *Raw/Text* mode, not those in *BlkRaw/BlkText/Command* mode
- **Protocol** is the communication protocol to use. Possible values are:
 - IPv4 : use the IPv4 connection protocol; if this is not possible then generate an error
 - IPv6 : use the IPv6 connection protocol; if this is not possible then generate an error
 - IP : use the IPv6 connection protocol; if this is not possible then use the IPv4 connection protocol.

The default is to inherit the protocol defined for the root object.

- **PublicCertData** has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- **PrivateKeyFile** has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- **PrivateKeyPass** has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- **PublicCertFile** has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- **PublicCertPass** has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- **PrivateKeyData** has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- **Priority** is the GnuTLS priority string (for complete documentation of this, see <http://www.gnutls.org/manual/gnutls.html#Priority-Strings>).
- **Magic** is a 32-bit integer used to check that the data has not been corrupted. Only valid for servers in *BlkRaw/BlkText* mode, not those in *Raw/Text/Command* mode.
- **X509** is a reference to an instance of the X509Cert class. Only valid when creating a secure server (see *Section 5.4*).

The [PublicCertData], [PrivateKeyFile], [PrivateKeyPass], [PublicCertFile], [PublicCertPass] and [PrivateKeyData] arguments are mutually exclusive with the [X509] argument.

Examples:

To create APLRPC, a *Command* mode server listening on port 5050:

```
DRC.Srv 'APLRPC' '' 5050 'Command'
0 APLRPC
```

To create a secure *Command* mode server on port 5050 of the local machine using the named certificate/key files and a TLS flag value of 64 (RequestClientCertificate):

```
cert<=>DRC.X509Cert.ReadCertFromFile 'path/server-cert.pem'
cert.KeyOrigin<='DER' 'path/server-key.pem'
certs<('X509' cert)('SSLValidation' 64)
DRC.Srv 'APLRPC' '' 5050 'Command',certs
0 APLRPC
```

To create a *Text* mode server (with an auto-generated name) listening on port 23, with a maximum buffer size of 1000 characters and a termination sequence of <CR>:

```
DRC.Srv '' '' 23 'Text' 1000 ('EOM' (␣UCS 13))
0 SRV00000000
```

Related function:

- `DRC.Clt` – see *Section A.5*

A.21 Function: DRC.Tree

Purpose: Returns information about the specified Conga object and all of its first generation children (or all existing Conga objects if the specified object is root).

Syntax: `rc tree ← DRC.Tree {objectname}`

where:

- `rc` is the return code (see *Section A.1*)
- `tree` is a 2-element vector in which the first element describes the specified object and the second element is a vector of trees describing each of its first-generation children (the second element is empty if the specified object has no children and it contains all existing Conga objects if the specified object is root).
- `objectname` is the name of the Conga object to describe.

For all objects, the description starts with the following elements:

- `[1]` is the name of the object
- `[2]` is the object's type (see *Section 4.1.1*)
- `[3]` is the object's state (see *Section 4.1.2*)

Some types of Conga object also include additional elements:

- Conga objects of type 0 (root):
 - [4] is the version of Conga
 - [5] is the thread count
- Conga objects of type 4 (commands) or 5 (messages):
 - [4] is the size of the object that has been processed so far (in bytes)
 - [5] is the size of the object that has not yet been processed (in bytes)

Example:

```

DRC.Srv 'S1' '' 5000
0 S1

DRC.Clt 'C1' 'localhost' 5000
0 C1

DRC.Wait 'S1' 100
0 S1.CON00000000 Connect 0

(rc (root subtree)) ← DRC.Tree '.'

]DISP root

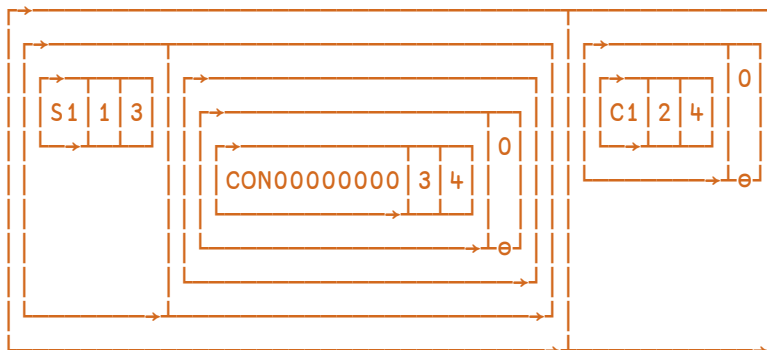
```



This elements in this result indicate that:

- [1] : The root object has no name (element is empty)
- [2] : The object type code is 0 (Root)
- [3] : The object state code is 2 (RootInit)
- [4] : The version number (and additional information) is available
- [5] : The number of semaphores currently in use for thread synchronisation is 1

]DISP subtree



The elements in this result indicate that there are two first-level children of the root:

- Conga object S1:
 - its object type code is 1 (server)
 - it is in state 3 (Listen)
 - it has a child object, CON00000000:
 - its object type is 3 (Connection)
 - it is in state 4 (Connected)
- Conga object C1:
 - its object type code is 2 (client)
 - it is in state 4 (Connected)

Similar functions:

- `DRC.Describe` – see *Section A.6*
- `DRC.Names` – see *Section A.14*

A.22 Function: DRC.Version

Purpose: Returns the Conga version number.

Syntax: `ver ← DRC.Version`

where:

- `ver` is a 3-element vector in which:
 - `[1]` is the major version number
 - `[2]` is the minor version number
 - `[3]` is the build number

Example:

```
DRC.Version
2 6 936
```

A.23 Function: DRC.Wait

Purpose: Waits for an event to occur.

Syntax: `rc objectname event data ← DRC.Wait`
`{clientname|servername|connectionname|commandname} [timeout]`

where:

- `rc` is the return code (a return code of 100 indicates a timeout, for other return codes see *Section A.1*)

- `objectname` is the name of the object on which the event occurred.
- `event` is the name of the event that occurred— see *Table A-3*
- `data` is the received data.
- `clientname|servername|connectionname|commandname` is the name of the object waiting for an event:
 - if a `servername` or `connectionname` name is specified, the `DRC.Wait` function will report events on the named object or any of its children.
 - If a *Command-mode* client is waiting on a specific command, the full `commandname` can be specified.
- `timeout` is the number of ms to wait before timing out. The default is 1,000.

Table A-3: *Types of event that can occur*

Event	Description
Block	<i>Text or Raw mode only:</i> A block of data was received and the connection is still open.
BlockLast	<i>Text or Raw mode only:</i> A block of data was received and the connection was closed; no more data is expected. If the connection is closed while it is inactive, a <code>BlockLast</code> event will be reported with empty data.
Connect	The Conga object has been created but has not yet participated in any data transmissions.
Error	An error occurred.
Progress	<i>Command-mode client only:</i> The server transmitted data using the <code>DRC.Progress</code> function.
Receive	<i>Command mode only:</i> Data has been received.

Examples:

```

      DRC.Srv 'S1' '' 5000
0 S1

      DRC.Clt 'C1' 'localhost' 5000
0 C1

      DRC.Wait 'S1' 5000
0 S1.CON00000000 Connect 0

      DRC.Send 'C1.fakename' 'Testing'
0 C1.fakename

```

```
DRC.Wait 'S1' 5000
0 S1.CON00000000.fakename Receive Testing
```



In *Command* mode, command names are carried over to the server.

Related functions:

- `DRC.Progress` – see *Section A.15*
- `DRC.Respond` – see *Section A.16*
- `DRC.Send` – see *Section A.17*

A.24 Class: DRC.X509Cert

Purpose: Provides a container to encapsulate X.509-style certificates. Dyalog Ltd recommends that this class is used when providing secure communications.

`#.DRC.X509Cert.[X509Cert]` is an instance of the X509Cert class.

Syntax – Shared Methods: These read certificates from various sources; they are not instance-specific.

To return certificate instances of all the certificates in a specified file:

```
certs ← DRC.X509Cert.ReadCertFromFile {filename}
```

To return certificate instances of all the certificates in a specified directory that match the specified pattern:

```
certs ← DRC.X509Cert.ReadCertFromFolder {pathname}
```

To return certificate instances of all the certificates in the specified certificate store:

```
certs ← DRC.X509Cert.ReadCertFromStore {storename}
```

where:

- `certs` is a vector of certificate instances where each element is of type `DRC.X509Cert`.
- `filename` is a certificate file name as a character vector, for example, `'server-cert.pem'`. Although it is a single file name, the file can contain multiple certificates.
- `pathname` is a character vector specifying the path to the directory that contains certificate files. It can be fully-qualified or relative to the current working directory. Wildcard characters can be used, for example, `'c:\mycerts*.pem'`, although if files match the pattern but are not valid certificate files then `certs` will be an empty vector.

- `storename` is a single certificate store name as a character vector (a list of all certificate store names is returned by `DRC.Certs 'ListMSStore'` – see *Section A.2*).

Syntax – Instance Methods: These act on specific certificate instances.

To verify whether the certificate is structurally valid:

```
bool ← #.DRC.X509Cert.[X509Cert].IsCert
```

To return the certificate chain for the certificate:

```
certs ← #.DRC.X509Cert.[X509Cert].Chain
```

To save the certificate instance to a file:

```
result ← [name] #.DRC.X509Cert.[X509Cert].Save path
```

To follow the certificate chain from the current certificate until a root certificate is found and, if possible, updates the `DRC.GetProp` function's `PeerCert` property so that the certificate chain is complete:

```
chain ← #.DRC.X509Cert.[X509Cert].CopyCertificateChainFromStore
```

where:

- `bool` is a Boolean indicating whether the certificate has a valid structure:
 - 0 : the certificate does not have a valid structure
 - 1 : the certificate has a valid structure
- `certs` is a vector of certificate instances where each element is of type `DRC.X509Cert`
- `result` indicates whether the file saved successfully:
 - 0 : the file saved successfully
 - `FALSE` : the file did not save successfully
- `name` is the name under which to save the file. If not specified, a name is built from the instance's Subject.
- `path` is a character vector specifying the path to the directory in which to save the certificate file. It can be fully-qualified or relative to the current working directory.
- `chain` is the number of certificates in the chain (including the calling instance and the root certificate).

Examples:

Verify whether `john` (an instance of the `X509Cert` class in the `Samples` namespace) is a structurally valid certificate:

```
Samples.john.IsCert
1
```

Return the issuer information for `john` (an instance of the `X509Cert` class in the `Samples` namespace):

```
(Samples.john.Chain).Formatted.Issuer
O=Test CA,CN=Test CA
```


A.24.1 Instances of the `DRC.X509Cert` Class

Each instance of the `DRC.X509Cert` class has the properties detailed in *Table A-4*.

Table A-4: *Properties of each instance of the `DRC.X509Cert` class*

Property	Description
<code>Cert</code>	Integer vector of raw certificate data.
<code>CertOrigin</code>	For certificates read from a certificate store this is: 'MSStore' storename For certificates read from a file this is: 'DER' and a fully qualified filename For example: 'DER' C:\apps\dialog141U64\TestCertificates\client\john-cert.pem
<code>Elements</code> <code>Extended</code> <code>Formatted</code>	<code>Elements</code> , <code>Extended</code> , and <code>Formatted</code> are namespaces that contain specific information about the certificate. <code>Elements</code> contains the information in a basic format, while <code>Formatted</code> and <code>Extended</code> have the same elements in a more human-readable format (<code>Extended</code> may, in some instances, contain additional information).

Table A-4: Properties of each instance of the *DRC.X509Cert* class (continued)

Property	Description
KeyOrigin	For keys read from a certificate store this is: 'MSStore' storename For keys read from a file this is: 'DER' and a fully qualified filename For example: 'DER' C:\apps\dialog141U64\TestCertificates\client\john-key.pem
LDRC	<i>Internal reference to the local DRC namespace.</i>
ParentCert	An instance of the certificate directly above this one in the certificate chain. Only relevant if this certificate is part of a certificate chain but not at the top of the chain.
UseMSStoreAPI	Boolean indicating which API to use to decode certificate information. Possible values are: <ul style="list-style-type: none"> 0 : Use the GnuTLS API 1 Use the Microsoft certificate store API  For applications that could be deployed on an operating system other than Microsoft Windows, the GnuTLS API should be used.

Not all certificates have values for all of the elements that are contained in the *Elements*, *Extended*, and *Formatted* properties, and some elements are more useful than others. *Table A-5* lists some of the more useful of the possible elements (it is not a comprehensive reference of X.509 certificate structure).

Table A-5: Some of the elements that can comprise the *Elements*, *Extended*, and *Formatted* properties of each instance of the *DRC.X509Cert* class

Property	Description
AlgorithmID	The cryptographic algorithm used to generate the signature, for example, RSA-SHA1 and DSA-SIGN.
Description	A text description of the certificate.

Table A-5: Some of the elements that can comprise the Elements, Extended, and Formatted properties of each instance of the DRC.X509Cert class (continued)

Property	Description
Issuer	The issuer of the certificate. Useful when validating certificate chains (the Issuer of a certificate should match the Subject of its parent certificate). Self-signed certificates have identical Issuer and Subject elements.
Key	The certificate's key in Boolean format.
KeyHex	The certificate's key in hexadecimal format.
KeyID	The certificate's key's cryptosystem, for example, RSA or DHE.
KeyLength	The length of the certificate's key (in bits). Maximum value is 16,384.
SerialNo	A number that uniquely identifies the certificate and is issued by the certification authority.
Subject	The subject of the certificate. Useful when validating certificate chains (the Subject of a certificate should match the Issuer of its child certificate). Self-signed certificates have identical Issuer and Subject elements.
ValidFrom ValidTo	Together, these two elements define the period of validity for the certificate.
Version	The version of the X.509 standard applied when creating the certificate (currently this is 3).

A.25 Operator: Samples.HTTPCmd

Purpose: Issues the specified HTTP command and waits for the result. Useful for interacting with RESTful web services.

Syntax: `rc rcvhdrs data peercert ← [certs] (cmd Samples.HTTPCmd) {url [params]}urlwithparams} [sendhdrs]`

where:

- `rc` is the return code (a return code of 100 indicates a timeout, for other return codes see *Section A.1*)

- `rcvhdrs` is a 2-column matrix of HTTP headers received from the server, in which:
 - `[;1]` is the header name, for example, `content-length`.
 - `[;2]` is the header value, for example, `8193`.
- `data` is the data received from the server.
- `peer cert` is the server's certificate. Only returned when interacting with a secure server (see *Section 5.4*).
- `certs` is the client's certificate (that is, the certificate for the machine calling the operator). Only valid when interacting with a secure server (see *Section 5.4*). Comprises `[X509 [SSLflags [priority]]]` where:
 - `X509` is a reference to an instance of the `X509Cert` class.
 - `SSL` is the sum of the relevant TLS flags (see *Appendix C*). Only valid when interacting with a secure server (see *Section 5.4*).
 - `priority` is the GnuTLS priority string (for complete documentation of this, see <http://www.gnutls.org/manual/gnutls.html#Priority-Strings>).
- `cmd` is the HTTP command to issue. Possible commands include 'GET', 'POST', 'DELETE' and 'PUT'; the specific commands that can be used are determined by the server.
- `url` is the URL to which the command is to be issued. Must be specified in the format `http[s]://www.abc.com`.
- `params` is a namespace or URL-encoded string specifying additional query parameters with which to filter the URL.
- `urlwithparams` is the URL to which the command is to be issued, specified in the format `http[s]://www.abc.com`, appended with a `?` character and a query string of `<name>=<value>` parameters separated by `&` characters.
- `sendhdrs` is any additional HTTP request headers. Specified as either a vector of 2-element (name-value) character vectors or a 2-column matrix of character vectors of names and values. For example, to be able to accept a gzip-compressed response, the server needs to be sent a header of `1 2p 'Accept-Encoding' 'gzip'`.

Examples:

For a GET command, parameters can be passed either as a part of the URL or separately. This means that:

```
url←'http://graphical.weather.gov/xml/sample_products/
browser_interface/ndfdBrowserClientByDay.php'
```

```
(params←[NS '']).(zipCodeList format numDays)←14586 '24+hourly' 7
```

```
('GET' Samples.HTTPCmd) url params
```

returns the same result as:

```
urlWithParams←'http://graphical.weather.gov/xml/sample_products/
browser_interface/ndfdBrowserClientByDay.php?zipCodeList=14586&
format=24+hourly&numDays=7'
```

```
('GET' Samples.HTTPCmd) urlWithParams
```

A.26 Function: Samples.HTTPGet

Purpose: Retrieves the contents of a web page from an internet site.

Syntax: `result ← [certs] Samples.HTTPGet {'url'}`

where:

- `result` comprises 3 elements:
 - the return code (see *Section A.1*)
 - HTTP headers returned as a 2-column matrix of attribute names and values. Browsers use this information to determine how to encode/decode data and provide other functionality to the end user
 - data returned as a character vector
- `certs` is the client's certificate (that is, the certificate for the machine calling the operator). Only valid when interacting with a secure server (see *Section 5.4*). Comprises [`X509` [`SSLflags` [`priority`]]] where:
 - `X509` is a reference to an instance of the `X509Cert` class.
 - `SSL` is the sum of the relevant TLS flags (see *Appendix C*). Only valid when interacting with a secure server (see *Section 5.4*).
 - `priority` is the GnuTLS priority string (for complete documentation of this, see <http://www.gnutls.org/manual/gnutls.html#Priority-Strings>).

If the server does not require a client certificate, then an empty left argument ' ' can be provided.

- `url` is a character vector specifying the complete URL of the web page whose contents are to be retrieved.

Example:

```
z←Samples.HTTPGet 'http://www.dyalog.com/news.htm'
```

Looking at each element of the result in turn:

```

1>z
0

2>z
http/1.1 200 ok
date          Tue, 03 Nov 2015 12:06:38 GMT
server        Apache/2.2.22 (Ubuntu)
x-powered-by  PHP/5.3.10-1ubuntu3.19
set-cookie    CMSSESSID7d7e905d=53n9dkaiquorpdsfp5sto29c3;
path=/
expires       Mon, 26 Jul 1997 05:00:00 GMT
cache-control no-store, no-cache, must-revalidate
last-modified Tue, 03 Nov 2015 12:06:38 GMT
cache-control post-check=0, pre-check=0
pragma        no-cache
vary          Accept-Encoding
transfer-encoding chunked
content-type  text/html; charset=utf-8

p3>z
16511

60↑3>z
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//E

```

If the `Content-Type` HTTP header specified `charset=utf-8`, then the `Samples.HTTPGet` function will perform the necessary decoding from UTF-8.

The `Samples.HTTPGet` function also supports the retrieval of web pages that have been protected using basic authentication; in this situation, the URL passed to the function must include the user ID and password information. For example:

```
result←Samples.HTTPGet 'http://user:pass@www.secret.com'
```

The `Samples.HTTPGet` function can retrieve secure web pages; the information that must be supplied to the function to enable it to make a secure connection depends on whether the server requires a client certificate:

- If the server requires a client certificate, then a left argument containing the `X509`, `SSLValidation` and `Priority` parameters for the connection must be provided (see *Section A.5* for a description of these parameters).
- If the server does not require a client certificate, then do one of the following:
 - provide an empty left argument `' '`.
 - prefix the URL with `https` rather than `http`.

When retrieving secure web pages, the result vector includes a fourth element comprising certificate information for the server.

A.27 Function: Samples.TestFTPClient

Purpose: Uses the `FTPClient` class to connect to ftp.mirror-service.org and accesses the file `pub/FreeBSD/README.TXT` from this website, counting and returning the number of characters in this file.

Syntax: `number ← Samples.TestFTPClient`

where:

- `number` is the count of characters in the file

Example:

```
Samples.TestFTPClient
pub/FreeBSD/README.TXT from ftp.mirror-service.org:

4259 characters read
```

A.28 Function: Samples.TestSecureWebClient

Purpose: Uses the `Samples.HTTPGet` function to test retrieval of the contents of the secure version of Dyalog Ltd's website (<https://www.dyalog.com/>).

Syntax: `result ← {certs} Samples.TestSecureWebClient {'url'}`

where:

`result` comprises 3 elements:

- the return code (see *Section A.1*)
- HTTP headers returned as a 2-column matrix of attribute names and values. Browsers use this information to determine how to encode/decode data and provide other functionality to the end user
- data returned as a character vector
- `certs` is certificate information:
 - if the server requires a client certificate, then a left argument containing the `X509`, `SSLValidation` and `Priority` parameters for the connection must be provided.
 - if the server does not require a client certificate, then an empty left argument `' '` can be provided.

- `url` is a character vector specifying the complete URL of the web page whose contents are to be retrieved.

Example:

```
z←Samples.TestSecureWebClient
'https://www.dyalog.com/news.htm'
```

Looking at each element of the result in turn:

```
1>z
0

2>z
http/1.1 200 ok
date          Tue, 03 Nov 2015 12:06:38 GMT
server        Apache/2.2.22 (Ubuntu)
x-powered-by   PHP/5.3.10-1ubuntu3.19
set-cookie    CMSSESSID7d7e905d=53n9dkaiqpuorpdfsf5sto29c3;
path=/
expires       Mon, 26 Jul 1997 05:00:00 GMT
cache-control  no-store, no-cache, must-revalidate
last-modified  Tue, 03 Nov 2015 12:06:38 GMT
cache-control  post-check=0, pre-check=0
pragma        no-cache
vary          Accept-Encoding
transfer-encoding chunked
content-type   text/html; charset=utf-8

p3>z
16511

60†3>z
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//E
```

If the `Content-Type` HTTP header specified `charset=utf-8`, then the `Samples.TestSecureWebClient` function will perform the necessary decoding from UTF-8.

The `Samples.TestSecureWebClient` function also supports the retrieval of web pages that have been protected using basic authentication; in this situation, the URL passed to the function must include the user ID and password information. For example:

```
result←Samples.TestSecureWebClient
'https://user:pass@www.secret.com'
```


A.29 Function: Samples.TestWebClient

Purpose: Uses the `Samples.HTTPGet` function to test retrieval of the contents of Dyalog Ltd's website (<http://www.dyalog.com/>).

Syntax: `result ← Samples.TestWebClient {'url'}`

where:

`result` comprises 3 elements:

- the return code (see *Section A.1*)
 - HTTP headers returned as a 2-column matrix of attribute names and values. Browsers use this information to determine how to encode/decode data and provide other functionality to the end user
 - data returned as a character vector
- `url` is a character vector specifying the complete URL of the web page whose contents are to be retrieved.

Example:

```
z←Samples.TestWebClient 'http://www.dyalog.com/news.htm'
```

Looking at each element of the result in turn:

```
1>z
0

2>z
http/1.1 200 ok
date          Tue, 03 Nov 2015 12:06:38 GMT
server        Apache/2.2.22 (Ubuntu)
x-powered-by   PHP/5.3.10-1ubuntu3.19
set-cookie    CMSSESSID7d7e905d=53n9dkaiqpurpdsfp5sto29c3;
path=/
expires       Mon, 26 Jul 1997 05:00:00 GMT
cache-control  no-store, no-cache, must-revalidate
last-modified  Tue, 03 Nov 2015 12:06:38 GMT
cache-control  post-check=0, pre-check=0
pragma        no-cache
vary          Accept-Encoding
transfer-encoding chunked
content-type   text/html; charset=utf-8

p3>z
16511
```

```
60†3>z
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//E
```

If the Content-Type HTTP header specified `charset=utf-8`, then the `Samples.TestWebClient` function will perform the necessary decoding from UTF-8.

The `Samples.TestWebClient` function also supports the retrieval of web pages that have been protected using basic authentication; in this situation, the URL passed to the function must include the user ID and password information. For example:

```
result←Samples.TestWebClient
'http://user:pass@www.secret.com'
```

A.30 Function: WebServer.Run

Purpose: Launches a web server.

Syntax: `rc ← WebServer.Run {path|functionname} {port} {servername}`

where:

- `rc` is the return code (see *Section A.1*)
- `path|functionname` is one of the following:
 - `path` – to present a flat set of HTML files as a webpage, specify the path to the root directory of the HTML files.
 - `functionname` – to execute APL for each page request, specify the name of the function in the active workspace that will intercept requests and manufacture output.
- `port` is the port number on which the server will listen
- `servername` is the name of the Conga server object to create

Example:

```
WebServer.Run 'c:\temp\htmlfiler\' 80 'http'
0
```

B Certificates

Many different file formats can be used for storing X.509 certificates, including PEM, DER, PFX, P7C and P12; the popularity of these formats varies between platforms. Conga supports the PEM and DER format files on all platforms; these have file extensions **.pem** and **.der** respectively. Certificates in other formats can be used after they have been converted to PEM or DER format files; this conversion can be performed using open source tools such as GnuTLS and OpenSSL (see the guide at <http://gagravarr.org/writing/openssl-certs/general.shtml> for information on converting between formats using OpenSSL).

B.1 PEM File Format

Files that have the PEM file format start with:

```
-----BEGIN CERTIFICATE-----
```

and end with:

```
-----END CERTIFICATE-----
```

They contain a base64-encoded version of the certificates and do not include any control characters.

The secure communications library GnuTLS (<http://www.gnu.org/software/gnutls/>) comes with a command line tool called **certtool** that can be used for creating certificates, keys and certificate requests as **.pem** files. It is documented at http://www.gnu.org/software/gnutls/manual/html_node/Invoking-certtool.html.

B.2 Generating Certificates and Keys



This is not supported on the AIX operating system.

The following example creates a set of certificates similar to the test certificates provided in the **[DIALOG]/TestCertificates** directory. This example is operating-system-independent and uses the GnuTLS open source secure communications library (see *Section B.1*).

The **CertTool** namespace includes code that checks whether **certtool.exe** exists in the location specified by **EXEC**. It then checks the version number of **certtool.exe** and signals a **DOMAIN ERROR** if it is less than 3.4.0 (version 3.4.0 introduced support for PKCS#7 and PKCS#12 encoded certificates).

To create a set of example certificates

1. Download and unzip the latest version of the GnuTLS secure communications library from <ftp://ftp.gnutls.org/gcrypt/gnutls/>.




Download and unzip the latest version of the GnuTLS secure communications library from your distribution's repository rather than using the above link.

2. Start a Dyalog Session and load the **CertTool** namespace into the workspace. For example:

```
]LOAD [DIALOG]/Samples/Certificates/CertTool.dyalog
```

3. Edit the values of the following names in the **CertTool** namespace:

EXEC	<p>(in the Init function, under the appropriate operating system)</p> <p>Fully-qualified path to the certtool.exe file in the bin directory of the unzipped GnuTLS secure communications library.</p> <p> The certtool.exe file is assumed to be on the path (this is true if it has been installed from the distribution's installation media or repositories).</p> <p>For example: <code>'c:\apps\gnutls-3.4.7\bin\certtool.exe'</code></p>
TARGET	<p>(in the Init function, under the appropriate operating system)</p> <p>Fully-qualified path to the directory in which to store the generated certificates and files.</p> <p>For example: <code>'c:\temp\TestCertificates\'</code></p>

SERIAL	<p>(in the <code>Init</code> function)</p> <p>Serial number of the first certificate generated. The other four certificates that are generated are assigned numbers related to this using the formula $SERIAL + 7 \times i - 1 + 15$ (the 7 can be changed in the <code>CommonAttr</code> function, line [3]).</p> <p>For example: 100</p>
C	<p>(in the <code>Init</code> function)</p> <p>Country in which the certificate is to be produced. This will be included in the subject information of the generated certificates.</p> <p>For example: UK</p>
O	<p>(in the <code>Init</code> function)</p> <p>Organisation that is producing the certificate. This will be included in the subject information of the generated certificates.</p> <p>For example: <code>DyalogLtd</code></p>
OU	<p>(in the <code>Init</code> function)</p> <p>Department within the Organisation that is producing the certificate. This will be included in the subject information of the generated certificates.</p> <p>For example: <code>Test</code></p>
ST	<p>(in the <code>Init</code> function)</p> <p>State/county/district in which the Organisation is located. This will be included in the subject information of the generated certificates.</p> <p>For example: <code>Hampshire</code></p>
CN	<p>(in the <code>Examples</code> function)</p> <p>The common name used on the certificate.</p> <p>For client certificates this is usually the name of a client or person. For example: <code>Ken Iverson</code></p> <p>For server certificates this is usually the DNS name of the server. For example: <code>www.dyalog.com</code></p>

- Run the `CertTool.Examples` function.
The directory specified by `TARGET` is created and populated with further directories and files.

The generated output is as follows:

- **CA** directory:
 - **ca-cert.pem**
The public certificate for the example CA. Used to authenticate client/server certificates.
 - **caconf.cfg**
Information about the CA certificate's properties.
 - **ca-key.pem**
The private key for the example CA. Used to sign client/server and CA certificates.
- **client** directory – four files for each `ClientCert` name defined in the `Examples` function. In this example, these are `John Doe` and `Jane Doe` (see lines [17] and [18] in the `Examples` function):
 - **<name>.cer**
A password-encrypted ASCII file containing the client's key and certificate in PKCS #7 file format.
 - **<name>.p12**
A binary file containing the client's key and certificate in PKCS #12 file format.
 - **<name>-cert.pem**
With **<name>-key.pem**, forms a client certificate's certificate/key pair.
 - **<name>-key.pem**
With **<name>-cert.pem**, forms a client certificate's certificate/key pair.
- **server** directory – four files for each `ServerCert` name defined in the `Examples` function. In this example, these are `localhost` and `myserver` (see lines [15] and [16] in the `Examples` function):
 - **<name>.cer**
A password-encrypted ASCII file containing the server's key and certificate in PKCS #7 file format.
 - **<name>.p12**
A binary file containing the server's key and certificate in PKCS #12 file format.
 - **<name>-cert.pem**
With **localhost-key.pem**, forms a server certificate's certificate/key pair.
 - **<name>-key.pem**
With **localhost-cert.pem**, forms a server certificate's certificate/key pair.

C TLS Flags

TLS flags are employed as part of the certificate checking process; they determine whether a secure client or server can connect with a peer that does not have a valid certificate.

The code numbers of the TLS flags described in *Table C-1* can be added together and passed to the `DRC.Clt / DRC.Srv` functions to control the certificate checking process. If you do not require any of these flags, then the `SSLValidation` parameter of these functions should be set to 0.

Table C-1: *TLS Flags*

Code	Name	Description
1	CertAcceptIfIssuerUnknown	Accept the peer certificate even if the issuer (root certificate) cannot be found.
2	CertAcceptIfSignerNotCA	Accept the peer certificate even if it has been signed by a certificate not in the trusted root certificates' directory.
4	CertAcceptIfNotActivated	Accept the peer certificate even if it is not yet valid (according to its <i>valid from</i> information).
8	CertAcceptIfExpired	Accept the peer certificate even if it has expired (according to its <i>valid to</i> information).
16	CertAcceptIfIncorrectHostName	Accept the peer certificate even if its hostname does not match the one it was trying to connect to.
32	CertAcceptWithoutValidating	Accept the peer certificate without checking it (useful if the certificate is to be checked manually – see <i>Section A.12</i>).

Table C-1: TLS Flags (continued)

Code	Name	Description
64	RequestClientCertificate	Only valid for a server; asks the client for a certificate but allows connections even if the client does not provide one.
128	RequireClientCertificate	Only valid for a server; asks the client for a certificate and refuses the connection if a valid certificate (subject to any other flags) is not provided by the client.

TLS flags have the same meanings for a server as for a client. However, for a server they are applied each time a new connection is established whereas for a client they are only applied when the client object is created.

D Conga Libraries

If an application that includes the conga workspace is shipped, then the relevant libraries will also need to be shipped. The libraries depend on the interpreter that is shipped with the application – *Table D-1-Table D-5* show the necessary libraries for each of the supported combinations of operating system, edition and width.

Table D-1: Libraries for interpreters on the AIX operating system

	Unicode Edition	Classic Edition
64-bit	conga27x64Uni.so libconga27ssl64.a	conga27x64.so libconga27ssl64.a
32-bit	conga27Uni.so libconga27ssl32.a	conga27.so libconga27ssl32.a

Table D-2: Libraries for interpreters on the Linux operating system

	Unicode Edition	Classic Edition
64-bit	conga27x64Uni.so libconga27ssl64.so	conga27x64.so libconga27ssl64.so
32-bit	conga27Uni.so libconga27ssl32.so	conga27.so libconga27ssl32.so

Table D-3: Libraries for interpreters on the macOS operating system

	Unicode Edition	Classic Edition
64-bit	conga27x64Uni.dylib libconga27ssl64.dylib	conga27x64.dylib libconga27ssl64.dylib
32-bit	conga27Uni.dylib libconga27ssl32.dylib	conga27.dylib libconga27ssl32.dylib

Table D-4: Libraries for interpreters on the Microsoft Windows operating system

	Unicode Edition	Classic Edition
64-bit	Conga27x64Uni.dll Conga27ssl64.dll	Conga27x64.dll Conga27ssl64.dll
32-bit	Conga27Uni.dll Conga27ssl32.dll	Conga27.dll Conga27ssl32.dll

Table D-5: Libraries for interpreters on the Raspberry Pi

	Unicode Edition	Classic Edition
64-bit	<i>n/a</i>	<i>n/a</i>
32-bit	conga27Uni.so	<i>n/a</i>

E Error Codes

Errors can be signalled at several levels within the Conga framework, including from the operating system, the Conga shared library or the GnuTLS library and within the APL coded portion of Conga.

If an error is generated when running Conga, then more information on that error can be obtained by entering:

```
DRC.Error {errorcode}
```

in the Dyalog Session.

Table E-1 details some of the errors that can be encountered and provides possible resolutions.

Table E-1: Possible error codes returned by Conga

Code	Source	Reason for Error	Possible Resolution
13	UNIX	Attempted to allocate a port with number less than 1025 without having root permission.	Either allocate a port number above 1024 or sign on as root (see 4001± in the <i>Dyalog APL Language Reference Guide</i>).
98	UNIX	Specified port number is already in use.	Allocate a different port number (it can take several minutes to de-allocate a port before it can be reused).
100	Conga	Timeout – nothing was received within the specified timeout period.	This is a normal occurrence and should be accommodated for in the client/server code.
1105	Conga	Could not receive data.	Re-establish the connection.

Table E-1: Possible error codes returned by Conga (continued)

Code	Source	Reason for Error	Possible Resolution
1119	Conga	Socket closed while receiving data (occurs when the connection is broken mid-block transfer).	Reconnect and resend the data.
1135	Conga	Maximum block size (as defined by the <code>BufferSize</code> parameter of the <code>DRC.Clt/DRC.Srv</code> function) exceeded when attempting to send/receive data.	Increase the value of the <code>BufferSize</code> parameter or chunk the data into smaller blocks.
1201	TLS	The handshake process that sets up a secure connection between the client and server before the certificates are exchanged is failing.	Ensure that the client and server are using the same encryption protocol and that both are using SSL/TLS.
1202	TLS	The certificate supplied by the peer is not valid.	Supply the TLS flag <code>CertAcceptWithoutValidating</code> to the <code>DRC.Srv/DRC.Clt</code> function (see <i>Appendix C</i>) to allow this connection and examine the certificate manually.
1203	TLS	One or more of the specified certificate files could not be loaded (either the file does not exist, it cannot be read or it is not a valid certificate file).	Ensure that the filenames being passed to the <code>DRC.Clt</code> and <code>DRC.Srv</code> functions are correct, that the files exist and that they are valid certificate files.
1204	TLS	There was an error setting up the TLS libraries.	Ensure that all GnuTLS files are present and valid.

F Change History

This appendix details the changes made at each version of Conga since the release of Conga version 2.0.

F.1 Version 3.1

Released with Dyalog version 17.0.

This version:

- changes the internal format of wide floating-point numbers from DPD to BID.

F.2 Version 3.0

Released with Dyalog version 16.0.

This version:

- adds:
 - two new server-only properties – FIFOMode and ConnectionOnly
 - integrated support for HTTP, including four new events (HTTPHeader, HTTPBody, HTTPChunk and HTTPTrailer), a new connection mode (Http), messages and utility libraries
 - a "Sent" event to enable the management of data buffers when sending large amounts of data
 - "Timeout" and "Closed" events (in addition to as error codes 100 and 1119)
 - dynamic loading of secure socket support
 - numerous new samples and utilities in a GitHub repository (<https://github.com/Dyalog/samples-conga>)
- adds support for:
 - multiple isolated Conga root objects

- WebSockets, enabling bi-directional asynchronous data transmission, including four new events (WSUpgrade, WSReceive, WSUpgradeReq and WSResponse) and a new connection mode (WebSocket)
- UDP (User Datagram Protocol) – *experimental*
- GnuTLS 3.5.16
- adds the ability to:
 - temporarily prevent new connections by setting the new Pause property
 - transmit files directly without having to first read the data into the workspace
 - permit or deny connections from specific address ranges
- simplifies configuration by establishing a default location for the Conga shared libraries
- removes or relocates seldom-used or atrophied samples

F.3 Version 2.7

Released with Dyalog version 15.0.

This version:

- adds a new namespace, `CertTools`; this can be used to generate certificates.
- removes the obsolete `TelnetServer` and `TelnetClient` classes from the `conga` workspace (the associated `TestTelnetServer` and `TestSecureTelnetServer` functions and `Parser` utility in the `Samples` namespace are also removed).
- merges the `WebServer.HttpsRun` method into the `WebServer.Run` method.
- allows an empty left argument to be supplied to the `Samples.HTTPGet` function.

F.4 Version 2.6

Released with Dyalog version 14.1.

This version:

- adds support for "blocked" raw and ASCII communications modes.
- adds a new `DRC.X509Cert.Save` method that saves the current certificate to file.
- adds a new strategy option (3) to the `DRC.GetProp` function's `ReadyStrategy` property; this selects the oldest connection but has improved performance over strategy option 2.
- adds new `HTTPCmd` operator and `HTTPPost` function to the `Samples` namespace.

- removes the need to specify protocol IPv4 on machines that do not support IPv6; in this situation, IPv4 will be selected by default.



In addition, this version:

- when using SSL/TLS, uses the Microsoft Windows "Trusted Root Certification Authorities" certificate store to verify system trust if the folder specified by the `DRC.GetProp` function's `RootCertDir` parameter contains no certificates
- adds a new `DRC.X509Cert.CopyCertificationChainFromStore` method that follows the certificate chain from the current certificate until a root certificate is found and, if possible, updates the `DRC.GetProp` function's `PeerCert` property so that the certificate chain is complete.

F.5 Version 2.5

Released with Dyalog version 14.0.

This version:

- incorporates a new version of the GnuTLS library to provide secure communications using SSL/TLS – this addresses a bug (CVE-2014-0092) whereby attackers could bypass the SSL/TLS protections.

F.6 Version 2.4

An internal update incorporating features in support of the Remote Interactive Development Environment (RIDE).

F.7 Version 2.3

Released with Dyalog version 13.2.

This version:

- adds a new `KeepAlive` property to the `DRC.GetProp` function; this causes a server to send periodic (heartbeat) messages to a client to determine whether the a connection is still live.



In addition, this version:

- now supports Integrated Windows Authentication (IWA), using the domain credentials of a Windows user for authentication. Two new functions, `DRC.ClientAuth` and `DRC.ServerAuth` provide client and server side IWA capabilities respectively.

F.8 Version 2.2

Released with Dyalog version 13.1.

This version:

- adds a new `DRC.Version` function that returns the current version of Conga.
- adds a new `DRC.flate` class that implements the deflate compression scheme (one of several content encoding schemes used by all major web servers and browsers to optimise the flow of data across networks) using the zlib open source compression library (for more information on zlib, see <http://zlib.net>).
- adds a new option, 2, to the `DRC.Send` function's `close` parameter; this sends a command without expecting a response. On the client side, the command is disposed of after sending. On the server side, the command is disposed of after receipt, thereby preventing the server from subsequently calling the `DRC.Respond` function.
- adds support for deflate HTTP compression in the `Samples.HTTPGet` function.
- enhances the `DRC.Describe` function to report the GnuTLS version.

F.9 Version 2.1

Released with Dyalog version 13.0.



Version 2.1 modifies how certificates are used to facilitate secure communications. Changes to the `DRC.Srv` and `DRC.Clt` functions when using certificates mean that Conga 2.0 applications that use certificates will require minor modification to use Conga 2.1.

This version:

- adds a new `DRC.X509Cert` class that encapsulates the structure and function necessary to use X.509 certificates with Conga. This is the recommended method for providing certificate information to the `DRC.Clt` and `DRC.Srv` functions.
- adds a new `DRC.Certs` function that provides the underlying functionality used by the `DRC.X509Cert` class to read and decode certificates.
- adds a new `PeerCert` property to the `DRC.GetProp` function; this returns an `X509Cert` object (certificate information).
- modifies the syntax used to pass certificate information to the `DRC.Srv` and `DRC.Clt` functions.
- adds a new strategy option (-1) to the `DRC.Init` function's `reset` parameter; this causes Conga to reload its underlying drivers.
- enhances the `Samples.HTTPGet` function to accept an `X509Cert` object as its (optional) left argument.

- enhances the `WebServer.HttpsRun` method to accept an `X509Cert` object argument.



In addition, this version:

- now reads/uses certificates located in Certificate Stores.

Index

C

CAs	See <i>Certificate authorities</i>
Certificate authorities	24
Certificate chains	31
Certificate revocation lists	26
Certificate stores	26
Classes	
DRC.X509Cert	65
Notation when calling	41
Compatibility with Dyalog	4
Conga object modes	10
BlkRaw mode	11
BlkText mode	11
Command mode	12
Raw mode	11
Text mode	10
Conga object names	6
Conga object properties	49
Conga object states	8
Conga object types	6
Client	7
Client-Server relationship	7
Command	7
Connection	7
Message	7
Root	6
Server	7
Conga objects	6
Conga workspace	33

D

DRC return codes	41
DRC.Certs (function)	42
DRC.ClientAuth (function)	42
DRC.Close (function)	43
DRC.Clt (function)	43
DRC.Describe (function)	46
DRC.Error (function)	47
DRC.Exists (function)	47
DRC.Flate.Deflate (method)	48
DRC.Flate.Inflate (method)	48
DRC.Flate.IsAvailable (method)	49
DRC.GetProp (function)	49
DRC.Init (function)	52
DRC.Names (function)	53
DRC.Progress (function)	54
DRC.Respond (function)	54
DRC.Send (function)	55
DRC.ServerAuth (function)	57
DRC.SetProp (function)	57
DRC.Srv (function)	59
DRC.Tree (function)	61
DRC.Version (function)	63
DRC.Wait (function)	63
DRC.X509Cert (class)	65

E

Error codes	85
Event types	64
Example namespaces	
RPCServer	36

Samples	34		
TODServer	39		
WebServer	35		
F		M	
Functions		Methods	
DRC.Certs	42	DRC.Flate.Deflate	48
DRC.ClientAuth	42	DRC.Flate.Inflate	48
DRC.Close	43	DRC.Flate.IsAvailable	49
DRC.Clt	43	Notation when calling	41
DRC.Describe	46	O	
DRC.Error	47	Operators	
DRC.Exists	47	Notation when calling	41
DRC.GetProp	49	Samples.HTTPCmd	69
DRC.Init	52	R	
DRC.Names	53	Return codes	41
DRC.Progress	54	S	
DRC.Respond	54	Samples.HTTPCmd (operator)	69
DRC.Send	55	Samples.HTTPGet (function)	71
DRC.ServerAuth	57	Samples.TestFTPClient (function)	73
DRC.SetProp	57	Samples.TestSecureWebClient	
DRC.Srv	59	(function)	73
DRC.Tree	61	Samples.TestWebClient (function) ..	75
DRC.Version	63	T	
DRC.Wait	63	TLS flags	81
Notation when calling	41	Troubleshooting	85
Samples.HTTPGet	71	W	
Samples.Test*	34	WebServer.Run	
Samples.TestFTPClient	73	Code outline	35
Samples.TestSecureWebClient	73	WebServer.Run (function)	76
Samples.TestWebClient	75		
WebServer.Run	76		
Code outline	35		
I			
Initialisation	4		
Installation	4		
L			
Libraries	83		