

REFERENCE MANUAL



Product and documentation by Aparajita Fishman

Copyright © 2001-2017 All rights reserved

Copyright and Trademarks

All trade names referenced in this document are the trademark or registered trademark of their respective holder.

Active4D is copyright Aparajita Fishman and Victory-Heart Productions.

4D and 4D Compiler are registered trademarks and 4D, 4D Server, and 4D Remote are trademarks of 4D, Inc.

Windows is a trademark of Microsoft Corporation.

Macintosh and macOS are trademarks of Apple Computer, Inc.

JavaScript and Java are trademarks of Sun Microsystems, Inc.

Acknowledgements

First of all, thanks to the makers of PHP for giving me the vision of a better way.

Thanks to the client who asked me to generate every single character of HTML in 4D code, which inspired me to come up with a better way.

Thanks to Mike Erickson for his convincing me that Active4D was worth doing. He was right.

Thanks to all of the users for their feedback and encouragement.

And thank you to David Adams for so kindly including a chapter about Active4D in "The 4D Web Companion" and for allowing me to include his HTTP chapter with these docs.

Table of Contents

Table of Contents	3
Introduction	25
What is Active4D?	25
HTTP Web Server.....	25
Server-Side.....	26
HTML-Embedded.....	26
Scripting Language.....	26
Development Environment.....	26
An Example	27
What Can Active4D Do?	27
Database and Protocol Support.....	28
A Brief History of Active4D	28
Installation	29
Plugin	29
Resource Files.....	29
Active4D Shells	30
Active4D Folder.....	30
Web Folder.....	31
Demo	31
Key Files	31
Key File Installation.....	31
Key File Info	32
Version Checking.....	32
License Types	33
Timeouts.....	33
Trial License.....	33
Deployment License.....	33
OEM License.....	34
Installation Options	34
Starting a Database from Scratch.....	34
Installing into a Non-Active4D Database.....	34
Updating an Existing Active4D 4.x/v5 Database.....	35
Installing the Predefined Session Handler.....	35
Post-Installation Configuration	36
Configuring for 4D's Web Server.....	37
Configuring for NTK.....	38
Configuring 4D Remote as a Web Server.....	39
Using the Pre- and Post-Execute Hooks	39
Pre-Execute Hook.....	40
Post-Execute Hook.....	40

- Configuration** 41
 - Config File Search Path** 41
 - Configuration Files** 42
 - The Default Directory 42
 - Path Format 43
 - The Standard Search Path and Path Lists 44
 - Active4D.ini 44
 - ExtensionMap.ini 47
 - Realms.ini 47
 - VirtualHosts.ini 47

- Security** 49
 - Source Code Security** 49
 - Web Server Security = Source Code Security 49
 - Circumventing Active4D 49
 - Potential Attacks** 50
 - Executing/Accessing Non-Web Files 50
 - The "safe script dirs" Config Option 50
 - Misusing Document Commands 51
 - The "safe doc dirs" Config Option 51
 - Spoofing Form Variables 51
 - The "auto create vars" Config Option 52
 - Uploading Huge Files 52

- HTTP Server** 53
 - What Is a Web Server?** 53
 - Active4D + Network Layer = Web Server 53
 - HTTP Fundamentals** 54
 - Active4D HTTP Request Handling** 54
 - Executable vs. Non-executable Files 54
 - Request Header Parsing 55
 - POST/PUT and File Upload Handling 56
 - Executable Request Handling 57
 - Non-Executable Request Handling 58
 - Configuration** 59
 - Active4D.ini 59
 - Cors.ini 59
 - ExtensionMap.ini 59
 - User Authentication** 59
 - Realms.ini 60
 - Virtual Hosting** 61
 - VirtualHosts.ini 61
 - A Virtual Host Example 63
 - HTTP Error Handling** 63
 - Customizing the Error Handling 64

- Invoking Active4D** 65
 - Types of Execution** 65
 - Request Execution** 65
 - A4D Execute <type> request Parameters 65
 - A4D Execute <type> request 66

A4D Execute BLOB request	69
A4D Execute 4D request	70
A4D Execute stream request	70
ReceiveCallback	71
..... Direct Execution	72
Uses for Direct Execution	72
A4D Execute file	72
A4D Execute text	73
A4D Execute BLOB	73
Interpreter	75
Flow of Execution	75
Embedding Source Code	75
Input Parsing	75
Language Syntax	77
English Only	77
Source Code Structure	77
Case Sensitivity	78
Expression-based	78
Comments	79
Identifiers	80
Custom Named Constants	81
Data Types	81
Compiler Declarations	81
Process/Interprocess Variables	82
Array Support	82
Pointer Support	83
Extended Boolean Expressions	83
Literals	83
String Literals	84
String Interpolation	84
Heredoc Strings	87
Date Literals	87
Time Literals	88
User-defined Constants	88
Typing of Values	88
Operators	88
Unary/Assignment Operators	88
Super Assign	89
In and Not In Operators	90
Regular Expression Operators	90
String Format Operators	91
Picture Operators	91
Pointer Dereference Operator	91
+ Operator	91
Character Reference Operator [[]]	92
Indexing Operator {}	92
Boolean Operator 	93
..... Control Structures	93
for each/end for each	94
break	94
return	94

- continue 94
- exit 94
- Examples 95
- Working with Paths** 96
 - URL-Style (Posix) Paths 96
 - Absolute vs. Relative Paths 96
 - Path Utilities 96
 - Path Limits 97
- Including Other Files** 97
 - Uses of Included Files 97
 - Including Only Once 98
- Calling 4D Methods** 98
 - Parameter Passing 98
 - Indirect Method Calls (aka Poor Man's method pointers) 99
- Collections** 99
 - Collection Handles 100
 - Local vs. Global Collections 100
 - Using Collections 100
 - Referencing Collection Values 101
 - Embedded Collections 101
 - Element Referencing 102
 - Iterating Over a Collection 102
- HTTP Data Access** 102
 - Request Data 103
 - _query and _form Collections 103
 - Testing Form Buttons 104
 - Response Data 104
- Working with Character Sets** 104
 - Platform Character Set 105
 - Output Character Set 106
 - Output Encoding 106
 - HTTP Request Decoding 107
 - Informing the Browser of Your Output Character Set 107
 - Working with Files 107
- Error Handling** 107
 - Script Timeout 108
- Methods** 109
 - Defining Methods** 109
 - Method Declaration** 109
 - Method Name 109
 - Method Parameter Declaration 110
 - Method Parameters** 110
 - Parameter Type 110
 - Scope 111
 - Referencing "Global" Local Variables 112
 - Pass by Reference 112
 - Default Parameters 113
 - Returning Values 114

Libraries	117
Library Search Path	117
Library Definition	118
Importing Libraries	118
Import Configuration	119
Import Errors	119
Automatic Re-Import	120
Library Namespace	120
Name Resolution	120
Library scope	121
The “global” library	121
Private methods	122
Library-private collections	122
Library Initialization/Deinitialization	123
Storing collections in library-private data	124
Creating a Poor Man’s Class	124
Limitations	126
Event Handlers	127
Event Handler Methods	127
On Application Start	127
On Request	128
On Authenticate	129
On Session Start	130
On Execute Start	131
On Execute End	131
On Session End	131
On Application End	132
Modifying the Active4D Library	132
Command Reference	133
4D Commands	133
Using a Default Table	133
Active4D Commands	136
Command Syntax	140
Unicode and Charsets	140
Arrays	141
{} (appending index)	142
{<index>} (from end index)	142
add element	143
append to array	143
ARRAY <type>	144
clear array	145
COPY ARRAY	145
Count in array	146
fill array	146
insert into array	147
is array	148
join array	148
multisort arrays	149
multisort named arrays	150

- resize array 150
- SELECTION/SELECTION RANGE TO ARRAY 150
- set array 151
- BLOBS** 152
- Collections** 153
 - collection 154
 - new collection 154
 - new local collection 155
 - new global collection 155
 - collection to blob 156
 - blob to collection 156
 - save collection 157
 - load collection 157
 - copy collection 158
 - deep copy collection 158
 - merge collections 159
 - clear collection 160
 - deep clear collection 160
 - get collection 161
 - get collection array 161
 - get collection array size 162
 - get collection item 163
 - get collection item count 163
 - get collection keys 164
 - set collection 164
 - set collection array 165
 - is a collection 166
 - collection has 166
 - count collection items 167
 - delete collection item 167
- Cryptography** 168
 - base64 decode 169
 - base64 encode 170
 - blowfish decrypt 171
 - blowfish encrypt 172
 - md5 sum 174
- Database** 175
 - Trigger Errors 175
 - Field name 176
 - get field numbers 176
 - get field pointer 177
 - QUERY 177
 - QUERY BY FORMULA 178
 - QUERY SELECTION 178
 - QUERY SELECTION BY FORMULA 179
 - SET QUERY DESTINATION 179
 - Table name 179
- Date and Time** 181
 - UTC Commands 181

Date	182
day of year	182
get utc delta	183
local datetime to utc	183
local time to utc	184
utc to local datetime	184
utc to local time	184
week of year	185
Debugging	186
current library name	187
current line number	187
current method name	187
get call chain	188
library list	188
write to console	189
Error Handling	190
get error page	191
get error status	191
get http error page	192
get log level	192
in error	192
log message	193
set error page	193
set http error page	194
set log level	194
File Uploads	195
How File Upload Works	195
Referencing File Uploads	195
The Importance of Filename Extensions	196
Upload Auto-Deletion	196
copy upload	196
count uploads	197
get upload content type	197
get upload encoding	197
get upload extension	198
get upload remote filename	198
get upload size	199
save upload to field	199
upload to blob	200
Form Variables	201
When Form Variables Are Query Params (and vice versa)	201
Posting JSON Data	201
Posting Raw Data	201
Multiple-choice Form Fields	201
_form	202
form variables	202
form variables has	202
get form variable	203
get form variable choices	203
get form variable count	204

- get form variables 205
- count form variables 205
- Globals** 206
 - Locking and Unlocking the Globals 206
 - globals 208
 - globals has 208
 - get global 208
 - get global array 209
 - get global array size 210
 - get global item 210
 - get global keys 211
 - set global 211
 - set global array 212
 - count globals 212
 - delete global 213
 - lock globals 213
 - unlock globals 213
- Iterators** 214
 - Using for each 214
 - Using Iterators 214
 - Iterator Validity 215
 - for each/end for each 216
 - more items 216
 - next item 216
 - get item key 217
 - get item value 217
 - get item type 217
 - get item array 218
 - is an iterator 218
- JSON** 219
 - new json 220
 - add to json 220
 - add datetime to json 222
 - add function to json 223
 - add rowset to json 224
 - add selection to json 226
 - start json array 230
 - end json array 231
 - start json object 231
 - end json object 232
 - json to text 233
 - write json 233
 - write jsonp 234
 - json encode 234
 - parse json 236
- Language** 238
 - call 4d method 239
 - call method 239
 - choose 240
 - define 242

EXECUTE.....	244
execute in 4d	244
for each/end for each	245
Get pointer	247
get throw code.....	248
get throw message.....	248
global	248
import	249
include	250
include into.....	250
longint to time.....	251
method exists.....	252
nil pointer	252
redirect	253
require.....	253
RESOLVE POINTER.....	254
sleep	254
throw.....	255
time to longint.....	255
Math	256
max of	257
min of	257
random between	257
ObjectTools	258
clear object	259
collection to object.....	259
object to collection.....	259
Pictures	260
Using the image.a4d Script	260
image.a4d (script file)	261
Loading from the 4D picture library.....	261
Loading from the database via query	262
Loading from the database via record number	262
Loading from a file	262
Loading from an Active4D method call	262
Loading from a 4D method call	263
Examples	263
write gif.....	264
write jpeg.....	265
write jpg.....	266
write png	266
Queries	267
QUERY/QUERY SELECTION.....	268
ORDER BY.....	268
ORDER BY FORMULA	269
Query Params	270
Query Params Items.....	270
Duplicate Query Parameters.....	270
_query.....	271

- query params 271
- query params has 271
- get query param 272
- get query param choices 272
- get query param count 273
- get query params 274
- count query params 275
- build query string 275
- Regular Expressions** 277
 - Pattern Syntax 277
 - Using Regular Expressions 277
 - regex callback replace 278
 - regex find all in array 279
 - regex find in array 280
 - regex match 281
 - regex match all 282
 - regex quote pattern 283
 - regex replace 284
 - regex split 287
- Request Cookies** 289
 - request cookies 290
 - get request cookie 290
 - get request cookies 291
 - count request cookies 291
- Request Info** 292
 - Request Info Collection Items 292
 - request info 293
 - get request info 293
 - get request infos 293
 - count request infos 294
- Request Value** 295
 - get request value 296
- Resources** 297
 - Get indexed string 298
 - STRING LIST TO ARRAY 299
- Response Buffer** 300
 - buffer size 301
 - response buffer size 301
 - clear buffer 301
 - clear response buffer 301
 - get response buffer 302
 - set response buffer 302
 - save output 303
 - end save output 304
 - set output charset 304
 - get output charset 305
 - set output encoding 305
 - get output encoding 306
 - write 307

write blob	307
writebr	309
writeln	309
writeln	310
write raw	310
=	311
Response Cookies	312
Cookie Fields	312
response cookies	313
get response cookie	313
get response cookies	314
set response cookie	314
set response cookie domain	315
get response cookie domain	316
set response cookie expires	316
get response cookie expires	316
set response cookie http only	317
get response cookie http only	317
set response cookie path	317
get response cookie path	318
set response cookie secure	318
get response cookie secure	318
count response cookies	319
delete response cookie	319
abandon response cookie	319
Response Headers	320
response headers	321
get response header	321
get response headers	321
set response header	322
count response headers	322
delete response header	323
Response Properties	324
get cache control	325
set cache control	325
get expires	325
set expires	326
get expires date	326
set expires date	326
get content type	327
set content type	327
get content charset	327
set content charset	328
get response status	328
set response status	328
Script Environment	329
_request	330
full requested url	330
current platform	330
get license info	331

get time remaining	332
get version	332
configuration	333
parameter mode	333
request query	334
set platform charset	334
get platform charset	335
set script timeout	335
get script timeout	336
set current script timeout	336
get current script timeout	336
Selecting Records	337
Loading Related Records	337
Configuring Related Record Auto-loading	337
Compatibility with Active4D 2.0.x	338
Examples	338
auto relate	340
ALL RECORDS, FIRST/LAST/NEXT/PREVIOUS RECORD	340
get auto relations	341
GOTO RECORD	341
GOTO SELECTED RECORD	342
Sessions	343
Session Lifetime	343
Session ID	344
Session Events	344
When Active4D Sends Session Cookies	345
Cookieless Sessions	345
Memory Caching of Sessions	346
Session Timeout and Memory Usage	346
Monitoring Memory Usage	346
Session Configuration	347
Session Handlers	348
session	350
session to blob	350
blob to session	351
get session	351
get session array	352
get session array size	352
get session item	353
get session keys	353
set session	354
set session array	355
session has	355
count session items	356
delete session item	356
abandon session	357
session id	357
session internal id	357
session local	358
session query	358
hide session field	359

set session timeout.	359
get session timeout	360
get session stats.	360
Strings	361
URL Encoding/Decoding	361
String Commands and Unicode.	361
% (formatting operator)	363
%% (formatting operator)	364
capitalize	365
cell	366
compare strings.	367
concat	368
Delete string	368
enclose	369
first not of.	370
first of	371
format string	371
identical strings	372
Insert string	372
interpolate string	373
last not of.	373
last of.	374
left trim	374
html encode	375
mac to html.	375
mac to utf8	376
param text	376
Position.	378
right trim	379
slice string	380
split string	381
String	382
Substring	383
trim	383
url decode	384
url decode path	384
url decode query	384
url encode	385
url encode path	385
url encode query	385
utf8 to mac	386
System Documents	387
Document Paths	387
Document Command Enhancements	387
Affected Commands	387
Error Codes	388
Working With Large Files	388
Append document	389
Create document	389
current file	390
current path	390

default directory	391
DELETE FOLDER	391
directory exists	392
directory of	392
directory separator	392
extension of	393
file exists	393
filename of	393
get root	394
join paths	394
MOVE DOCUMENT	395
native to url path	395
Open document	396
RECEIVE PACKET	396
requested url	397
resolve path	397
SEND PACKET	398
SET DOCUMENT POSITION	398
split path	398
url to native path	399
Timestamps	400
Timestamp Format	400
Timestamp Time	400
Timestamp Normalization	400
Using Timestamps with Optimistic Locking	401
timestamp	403
add to timestamp	404
timestamp difference	404
timestamp string	405
timestamp date	405
timestamp time	406
get timestamp datetime	406
timestamp year	407
timestamp month	407
timestamp day	407
timestamp hour	408
timestamp minute	408
timestamp second	409
timestamp millisecond	409
User Authentication	410
auth password	411
auth type	411
auth user	411
authenticate	412
current realm	412
Variables	413
defined	414
get local	414
local variables	415
set local	415

type descriptor	416
undefined	416
variable name	417
Plugin Commands	418
A4D Abandon session	419
A4D Base64 decode	419
A4D Base64 encode	419
A4D Blowfish decrypt	420
A4D Blowfish encrypt	420
A4D FLUSH LIBRARY	420
A4D Get IP address	421
A4D Get MAC address	422
A4D Get root	422
A4D GET SESSION DATA	422
A4D GET SESSION STATS	423
A4D GET LICENSE INFO	424
A4D Get MAC address	424
A4D Get time remaining	424
A4D Get version	425
A4D Import library	425
A4D LOG MESSAGE	425
A4D MD5	426
A4D Native to URL path	426
A4D RESTART SERVER	426
A4D Set HTTP body callback	427
A4D SET ROOT	427
A4D STRIP 4D TAGS	428
A4D URL decode path	428
A4D URL decode query	428
A4D URL encode path	429
A4D URL encode query	429
A4D URL to native path	429
Standard Libraries	431
Using the Standard Libraries	431
a4d.console	432
clear	433
dump array	433
dump collection	434
dump form variables	435
dump license info	435
dump query params	435
dump request info	436
dump session	436
a4d.debug	437
dump array	438
dump collection	439
dump configuration	440
dump form variables	440
dump license info	440

- dump locals 441
- dump query params 441
- dump request 442
- dump request info 442
- dump selection 442
- dump session 443
- dump session stats 444
- a4d.json** 445
 - new 446
 - add 446
 - addArray 449
 - addDateTime 450
 - addFunction 451
 - addRowSet 452
 - addSelection 455
 - startArray 458
 - endArray 459
 - startObject 459
 - endObject 460
 - toJSON 461
 - write 462
 - writeln 462
 - encode 463
 - encodeArray 463
 - encodeBoolean 464
 - encodeCollection 464
 - encodeDate 466
 - encodeString 466
 - parse 467
 - convertJSONDates 468
- a4d.lists** 470
 - append 471
 - arrayToList 471
 - changeDelims 472
 - contains 472
 - containsNoCase 473
 - deleteAt 473
 - find 473
 - findNoCase 474
 - first 474
 - getAt 474
 - insertAt 475
 - last 475
 - len 476
 - listToArray 476
 - prepend 476
 - qualify 477
 - rest 477
 - setAt 478
 - sort 478
 - valueCount 479

valueCountNoCase.....	479
valueList.....	479
a4d.utils	481
applyToSelection.....	482
articleFor.....	482
blobToCollection.....	483
blobToSession.....	483
camelCaseText.....	484
chopText.....	484
collectionToBlob.....	485
cud.....	486
deleteSelection.....	491
filterCollection.....	492
formatUSPhone.....	492
getMailMethod.....	493
getPictureDescriptor.....	493
getPointerReferent.....	494
getSMTPAuthorization.....	494
getSMTPAuthPassword.....	495
getSMTPAuthUser.....	495
getSMTPHost.....	495
nextID.....	496
ordinalOf.....	496
parseConfig.....	497
reverseArray.....	499
selectionRangeToCollection.....	499
selectionToCollection.....	499
sendMail.....	501
sessionToBlob.....	503
setMailMethod.....	503
setSMTPAuthorization.....	504
setSMTPHost.....	504
truncateText.....	504
unlockAndLoad.....	505
validPrice.....	506
yearMonthDay.....	507
a4d.web	508
br.....	509
buildOptionsFromArrays.....	509
buildOptionsFromLists.....	510
buildOptionsFromOptionArray.....	511
buildOptionsFromOptionList.....	512
buildOptionsFromRowSet.....	513
buildOptionsFromSelection.....	514
checkSession.....	515
checkboxState.....	515
collectionItemsToQuery.....	516
collectionToQuery.....	517
embedCollection.....	517
embedCollectionItems.....	519
embedFormVariableList.....	519

embedFormVariables	519
embedQueryParams	520
embedVariables	520
emptyTag	521
formVariableListToQuery	521
getEmptyFields	522
getUniqueID	522
getVariablesIterator	522
hideField	523
hideUniqueField	523
saveFormToSession	523
validateTextFields	524
validEmailAddress	524
warnInvalidField	525
writeBold	525
Batch	526
Batch Attributes	526
Creating a Batch	526
How Batches Are Calculated	527
Generating Batch Links	528
Iterating Through Rows	531
dumpDefaults	532
getDefaults	532
getStarts	533
makeFuseboxLinks	533
makeLinks	534
new	537
newFromArray	538
newFromRowSet	538
newFromSelection	539
next	540
previous	540
setDefault	541
Breadcrumbs	542
Using Breadcrumbs	542
Customizing Breadcrumbs Appearance	544
add	545
dumpLib	545
fuseboxNew	546
new	546
setDivId	547
setSeparator	547
write	547
fusebox	548
An Overview of Fusebox	548
Why Should I Use Fusebox?	548
How Do I Learn Fusebox?	549
Active4D's Fusebox Implementation	549
Configuring Fusebox	550
core	551

getURLFactory	551
handleError	551
handleErrorInline	552
invalidAction	553
isFuseboxRequest	553
makeURL	554
postHandleError	554
sendFuseaction	555
setURLFactory	555
fusebox.head	557
How To Use This Library	557
addCSS	558
addDumpStyles	559
addJavascript	560
addJS	560
addMetaTag	562
getTitle	562
setTitle	563
write	563
RowSet	564
Enter the RowSet	564
Using RowSets	566
Subsetting Source Rows	568
RowSet Cursors	568
Persistent RowSets	568
Which RowSet to Use	570
afterLast	571
beforeFirst	571
clearPersistent	571
columnCount	572
currentRow	572
dumpPersistent	572
dump	573
findColumn	573
findRow	573
first	574
getColumn	574
getData	575
getEnd	575
getPersistentList	575
getRow	576
getStart	576
getTimeout	576
gotoRow	577
isAfterLast	577
isBeforeFirst	578
isFirst	578
isLast	578
last	579
maxRows	579
move	579

newFromArrays	580
newFromCachedSelection.....	581
newFromData.....	582
newFromFile.....	583
newFromSelection	584
next	589
persistent.....	589
previous	589
rowCount.....	590
setColumnArray.....	590
setColumnData	591
setRelateOne	591
setTimeout	592
sort	592
sourceRowCount.....	593
timedOut	593
SessionHandler	594
nextId	595
read	595
write.....	596
delete	596
purge.....	597
Debugging	599
The Basics	599
Using write	599
Tracing execution.....	599
Standard Library Methods	600
a4d.console and a4d.debug	600
The Active4D Log	601
Changing the Log Level	602
The Session Editor.....	603
Using the Session Editor.....	603
The Session Monitor	604
Displaying the Session Monitor.....	604
Using the Session Monitor.....	605
The Active4D Debugging Console	606
Using the Debugging Console.....	606
Filtering Console Messages.....	607
Tuning Console Refresh	608
Error Handling	609
The Default Error Message	609
Using a Custom Error Page	610
The "error page" Option	610
The "set error page" Command	610
Error Page Variables.....	611
Custom Error Handling in Fusebox.....	612
The Active4D Log	613
Log Levels	613

Index of Commands 615

ISO Language Codes 629

Named Constants..... 631

 Grouped by Function..... 631

 633

 Alphabetical 633

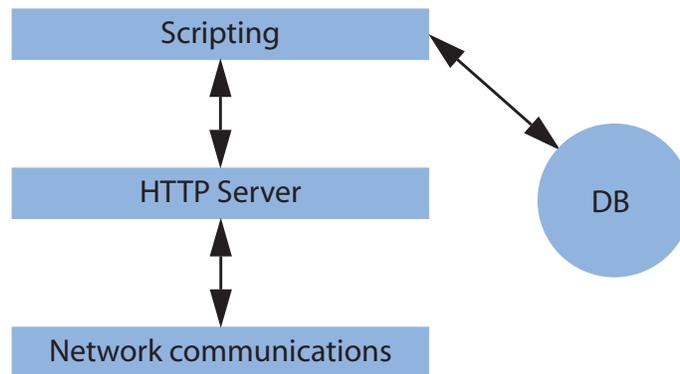
CHAPTER 1

Introduction

Welcome to Active4D! You have chosen the ultimate environment for building world-class web applications with 4D.

What is Active4D?

A dynamic web scripting environment has four layers: a TCP communications layer, an HTTP server layer, a scripting layer, and a database layer. The different layers can be pictured like this:



The HTTP server builds on top of the network layer, and the scripting layer builds on top of the HTTP server. The database is in a separate, independent layer.

Active4D is both an HTTP web server and a server-side HTML-embedded scripting language and development environment.

That's a mouthful. Let's break down the sentence and look at what it means.

HTTP Web Server

An HTTP (HyperText Transfer Protocol) web server is a piece of software that receives HTTP requests, processes those requests and sends an appropriate response to the origin. In essence, this is what a web server does.

4D's built-in web server handles TCP communications and handles most of the HTTP protocol.

NTK (Network Toolkit) is just what it says — not an HTTP server, but a toolkit that *allows* you to write an HTTP server. Both come with sample databases which implement a web server well for most purposes.

Active4D implements an HTTP server in a plugin. As an HTTP server, it offers:

- **Ease of setup:** Active4D is ready to run out of the box. There is no complicated setup needed.
- **Speed:** As a plugin, Active4D always runs at native compiled speed, even in an interpreted database. This can mean huge gains in productivity during development.
- **Features:** Active4D adds a wealth of advanced features like Virtual Hosting that offer you some of the benefits of dedicated web servers like WebStar.

Server-Side

Scripting languages like JavaScript download their source code to the client browser and execute it there. Active4D, on the other hand, is executed on the server and the source code is removed before the page is sent to the client.

HTML-Embedded

With Active4D, dynamic HTML generation is directly embedded inside the HTML page.

4D's web tag system, while providing some of the benefits of embedded scripting, still requires you to write many, many 4D methods to handle simple tasks like queries and ordering. In Active4D this is handled directly within the web page. In fact, with Active4D all of your application code can (and in most cases should) exist entirely outside of 4D itself.

Scripting Language

To generate dynamic HTML you must have a programming language. Active4D is a full 4D interpreter in a plugin. To program in Active4D you don't need to learn a new syntax or specialized tag language. In many cases you can literally copy 4D code from a method and paste it into a web page to be executed by Active4D.

Active4D supports all 4D data types except for 2D arrays. It implements over 170 of the most important 4D commands, as well as almost 300 new commands which provide unmatched power — and ease — to your web development.

Active4D also adds many often-requested extensions to the language, such as the **break** and **return** keywords, pass-by-reference, sophisticated string formatting, and associative arrays.

Development Environment

Active4D comes with a plethora of built-in debugging tools, both on the client side and on the 4D side. In conjunction with all of the other features Active4D provides, there simply is no more powerful or more productive web development environment for 4D, period. Nothing else comes close.

An Example

Here's a simple example of what an Active4D page looks like:

```
<html>
  <body>
    Here are 5 good reasons to use Active4D:<br />
    <%
      for ($i; 1; 5)
        writebr("It rocks!")
      end for
    %>
  </body>
</html>
```

Note that the code is fully embedded in the page, and that except for the **writebr** command, which writes an expression to the HTML page, the embedded code *is* 4D code.

During execution, Active4D passes HTML through to the response, executes everything inside the `<% %>` tags, and spits out the following:

```
<html>
  <body>
    Here are 5 good reasons to use Active4D:<br />
    It rocks!<br />
  </body>
</html>
```

The source code has been replaced with the output of the Active4D code. You can find out more about programming with Active4D in Chapter 7, "Interpreter."

What Can Active4D Do?

Active4D can do anything that 4D can do in the context of a web page, plus quite a few tricks that would be time-consuming if not impossible to implement in 4D.

If Active4D's commands don't fit the bill, you can write methods and libraries of methods completely within Active4D, using plain text files. And if you need to use a command or plugin that is not in Active4D's language, you can either execute it within Active4D using the **execute in 4d** command, or you can call any 4D method, passing parameters of any type and receiving a result of any type.

Primarily, however, Active4D is designed to act as a conduit between the database and the web page. As such, it excels at the following essential tasks:

- Collecting information from forms and queries for easy access within your web application

- Managing user sessions
- Querying and manipulating data without having to write specialized 4D methods
- Handling file uploads
- Formatting text for output to HTML
- Generating JSON data for use with Javascript client libraries
- Handling character set issues

Database and Protocol Support

Active4D comes with built-in support for 4D's built-in database engine. You could, without much work, write wrapper methods for the various connectivity plugins that would allow you to work with other databases such as MySQL.

In addition, you could also easily write method wrappers around the 4D Internet Commands to allow you to access other internet protocols like SMTP.

A Brief History of Active4D

Many years ago I had the occasion to use another dynamic embedded scripting language: PHP. I was amazed at the power it provided, and more importantly I realized the advantages of the embedded scripting model.

A little later I was hired to work on a vertical market, web-based application that used 4D. I was asked to add a new module to the existing application. To my amazement, this application generated every single character of HTML programmatically in 4D methods. This was before 4D offered its web tag system.

Having been exposed to embedded scripting, I quickly saw that this was a fundamentally flawed approach, and I said to myself, "There has to be a better way." So I set about creating one.

I never intended to create Active4D — it just sort of happened. A few months after I began with a simple 4D-based parser that replaced variable and field names, I had a full working 4D interpreter in a plugin.

A few features and a few months later, Active4D 1.0 was released. That was November of 2000. Since then I have had a chance to use Active4D heavily in web projects, and decided it was time to fill in all of the holes in its feature set.

I systematically went down the list of every important feature in ASP (Microsoft's Active Server Pages) and PHP, and implemented each and every one. And wherever possible I added a few features that they don't have.

The result is what you see here: Active4D v6.4, the *ultimate* 4D web environment.

CHAPTER 2

Installation

Installation of Active4D is a fairly simple matter and should take no more than a few minutes.

First you must of course download the latest version from:

<http://www.aparajitaworld.com/downloads/active4d/latest.zip>

There are four elements to Active4D:

- **Plugin:** There is a single plugin for all platforms.
- **Shell:** You must choose the appropriate shell for the environment in which you are running Active4D. The shell acts as a conduit between the network communications layer and Active4D.
- **Documentation:** Your best friend. :-)
- **Demo:** This 4D v13 database and web site demonstrates many of the key techniques you will use with Active4D.

Plugin

The Active4D plugin bundle contains code for macOS (Intel) and Windows, both in 32/64-bit. This allows you to run on any platform or architecture.

Resource Files

Within the Active4D plugin bundle are resource files used by Active4D. These resource files are located in Active4D.bundle/Contents/Resources. The resource files are:

- **Active4D_58l.dat:** Contains resources specific to Active4D.
- **icudt58l.dat:** Contains resources used by ICU, a code library used by Active4D. This is a very large file because it contains Unicode and internationalization data for every country and language in the world.

The default location for these files is within the plugin bundle. They may also be placed in the `<shared 4D folder>/com.aparajita/icu` folder. The shared 4D folder is the parent of the folder which is returned by **Get 4D folder(Licenses folder)** within 4D. For the location of this folder, please refer to the 4D documentation for the **Get 4D folder** command.

If you decided to use the shared “icu” folder, *both* resource files must be placed there.

Active4D Shells

The archive contains two 4D database folders which contain Active4D *shells* — the code to integrate Active4D with either 4D's web server or NTK. In addition, the shells contain the forms and methods which implement the server-side debugging features of Active4D.

Warning: You must use NTK 3.1 or later with the NTK shell in Active4D v6.4 or later.

Within each database folder, in addition to the standard 4D files and folders, you will also see:

- An "Active4D" folder with the Active4D standard libraries and configuration files. For more information see "Active4D Folder" on page 30.
- A "web" folder with Active4D utility scripts. For more information see "Web Folder" on page 31.

Active4D Folder

This folder contains the *Active4D standard libraries* and *config files* which are used by Active4D.

- **a4d.console.a4l:** A library which provides methods for dumping various debugging information to the debug console.
- **a4d.debug.a4l:** A library which provides methods for dumping various debugging information to a web page.
- **a4d.json.a4l:** A library which provides a full suite of methods for JSON (JavaScript Object Notation) data for use with Javascript client libraries.
- **a4d.lists.a4l:** A library which provides a full suite of methods for working with delimited lists.
- **a4d.utils.a4l:** A library which provides various non-web-related utilities.
- **a4d.web.a4l:** A library which provides various web-related utilities, mostly for working with forms.
- **Active4D.a4l:** A special library that contains global *event handlers* for your application. For more information on event handlers, see Chapter 10, "Event Handlers."
- **Active4D.ini:** A config file which sets most of the options in Active4D.
- **Batch.a4l:** A library which provides methods for splitting large query results into batches of rows and creating links to those batches.
- **Breadcrumbs.a4l:** A library which provides methods for creating a "trail" of "breadcrumbs" — links to other pages — to aid the user in navigating through your site.
- **ExtensionMap.ini:** A config file which maps filename extensions to Macintosh file types and MIME types.
- **fusebox.a4l:** A library which implements the Fusebox 3 core.

- **fusebox.conf.a4l:** A library in which you can configure fusebox.
- **Realms.ini:** A config file which maps portions of a URL to security realms.
- **RowSet.a4l:** A library which provides an implementation independent representation of rows of data.
- **VirtualHosts.ini:** A config file which provides the routing table for virtual hosts. For more information on virtual hosts, see “Virtual Hosting” on page 61.

For more information on the standard libraries, see Chapter 12, “Standard Libraries.” For more information on config files, see Chapter 3, “Configuration.”

Web Folder

This folder contains files that you may use in your site:

- **image.a4d:** This file is an all-purpose script for dynamically loading images from the database or from disk and optionally creating thumbnails from them. For more information, see “image.a4d (script file)” on page 261.
- **sed.a4d:** This file is a reusable script for examining and editing sessions. For more information, see “The Session Editor” on page 603.

Demo

The demo is a 4D v13 database and web site that illustrates many key Active4D techniques. To run the demo do this:

- 1 Open the demo database. You can safely open and convert it in later versions of 4D.
- 2 Open a browser and enter “http://localhost:8080” or “http://127.0.0.1:8080”.
- 3 Explore the demo and look at the underlying code.

Key Files

Active4D runs in one of several modes, depending on the license currently in force. The license is determined by a *key file* (or lack thereof), which contains information about you and your license.

Warning: You should always keep a backup of your key file.

Key File Installation

A key file must have a name which begins with “Active4D.” and ends with “.key” (the dot in the two parts may be the same character). Active4D will look for a key file in several places, in the following order:

- 1 The `<user 4D>/com.aparajita/Active4D` folder. The user 4D folder is the folder which is returned by **Get 4D folder(Active 4D Folder)** within 4D. For the location of this folder, please refer to the 4D documentation for the **Get 4D folder** command.

- 2 The `<4D shared licenses>/com.aparajita/Active4D` folder. The 4D shared licenses folder is the folder which is returned by **Get 4D folder(Licenses Folder)** within 4D. For the location of this folder, please refer to the 4D documentation for the **Get 4D folder** command.
- 3 The `Active4D.bundle/Contents/Resources` folder.

Warning: If you are running 4D Server as a service, the key file must be placed in the `4D licenses/com.aparajita/Active4D` folder (location #1 above).

Note: To place the key file in the `Active4D.bundle/Contents/Resources` folder on macOS, right-click on the `Active4D.bundle` plugin and select “Show Package Contents”. Then navigate into the `Resources` folder.

If you have a key file in more than one place the first one found will take precedence.

Key File Info

Key files contain the following information in encrypted format:

- The name and company of the key file purchaser
- The license type
- The platform for which the key is licensed
- The IP address for which the key is licensed (only used with deployment licenses)
- The expiration date of the key’s license (keys never expire, only their licenses do)
- An encoded serial number

You can access this information both through a 4D plugin call (**A4D GET LICENSE INFO**) and through the Active4D command **get license info**. Even easier, you can use the **A4D_Prefs** method in 4D to display a dialog with the license info, or use the **a4d.debug.dump license info** method to display license info in a web page or the **a4d.console.dump license info** method to display license info in the Active4D console.

Version Checking

Encoded in the serial number is the major/minor version of Active4D for which the key is licensed. If you attempt to use an otherwise valid key file with a version of 4D newer than the version encoded in the key file, the license will be temporarily downgraded.

Note: Version checking does not apply to bug fix releases; i.e. versions 6.4 and 6.4.1 are considered the same version, whereas 6.4 is considered newer.

License Types

Each license has certain limitations of which you should be aware. The license types Active4D supports are:

License	Limits
Trial	Times out after 8 hours of continuous use
Deployment	Tied to a single IP address, never times out on that address, times out after one week on another address
OEM	No time or IP address limitation, requires special 4D code which ties it to a 4D structure

Timeouts

The trial license has a timeout period. When 4D has run continuously for the timeout period, Active4D becomes disabled. Thereafter any attempts to execute Active4D will return the text "Active4D has exceeded its time limit." At that point 4D must be restarted to reset the timeout and enable Active4D.

Trial License

If Active4D cannot find a key file, or if the key file is incorrect in some way, Active4D operates in trial mode.

Deployment License

If Active4D finds a key file whose license type is Deployment, Active4D checks the IP address of the host machine against the IP address in the key file. If an exact match is found, Active4D operates in deployment mode with no timeout, unless the version check fails, in which case Active4D reverts to trial mode.

If the host machine has more than one IP address, Active4D will check all of the host's IP addresses for a match. On macOS, only interfaces whose names begin with "en" (ethernet interfaces) will be checked.

If an IP address match cannot be made, Active4D reverts to expired mode, which times out after one week of continuous use. Until you get a new key file, you must restart 4D once a week.

Ideally you should obtain a new key file in advance if you know your application will be moved to a server with a different IP address. However, if you are unable to secure a new key file before switching, this scheme allows you to operate Active4D for a full week after switching. During this time you should be able to obtain a new key file. Once you have a new key file, you must restart 4D to have Active4D run in Deployment mode once more.

If you are running Active4D on 4D Server, checking the license status from a 4D Remote will indicate the license is running in expired mode, since the IP address of the 4D Remote cannot match the Server IP address.

To check the license status of a Server-based key file, you must use the **a4d.debug.dump license info** method, the **a4d.console.dump license info** method, or the **get license info** command on a page served by Active4D running on Server.

OEM License

For information on OEM licensing, please contact sales@aparajitaworld.com.

Installation Options

There are several installation scenarios: starting a database from scratch, installing into a non-Active4D database, and updating an existing Active4D 4.x database.

Starting a Database from Scratch

The Active4D shell databases are perfect for use when you are starting a database from scratch, as they are ready to run. You just need to build your database on top of the shell appropriate to network layer you plan to use (4D/NTK).

Installing into a Non-Active4D Database

The Active4D environment relies on a small set of 4D methods, lists, forms and styles, all of which have the prefix "A4D_" to prevent name conflicts with existing code (I hope).

To install Active4D into an existing database, follow these steps:

- 1 Open the shell database corresponding to the network layer you will be using (4D/NTK).
- 2 Open the Explorer and click "Home" on the left side of the Explorer window.
- 3 Open the target database in a second copy of 4D.
- 4 Open the Explorer in the target database and click "Home" on the left side of the Explorer window.
- 5 Arrange the Explorer windows in the shell and target databases so the contents lists are visible in both.
- 6 From the shell database, drag the "Active4D" folder from the Explorer window to the contents list of the Explorer window in the target database.
- 7 Switch to the target database. You may see a warning dialog asking you if you want to continue. Click OK.
- 8 A "Moving Dialog" dialog will appear. Click Next, then click OK.
- 9 Quit the copy of 4D running the shell database.
- 10 Copy the Active4D plugin bundle into a directory which is accessible to the target database. Please consult the 4D documentation for information on where plugins may be placed.
- 11 Open the shell database folder. On macOS, you must right-click on the database and select "Show Package Contents".
- 12 Open the target database folder.

- 13 Copy the “Active4D” and “web” folders from the shell database folder to the target database folder.
- 14 Open the target database in 4D.
- 15 In your startup sequence (*On Startup*, *On Server Startup*, or a method called by them), add a call to the method *A4D_Startup*.
- 16 In your shutdown sequence (*On Exit*, *On Server Shutdown*, or a method called by them), add a call to the method *A4D_Shutdown*.
- 17 If you want to access the Active4D preferences dialog, console, or session monitor, you will need to add menu items that call *A4D_Prefs*, *A4D_Console* or *A4D_SessionMonitor* respectively.
- 18 If you plan on using 4D Remote as a web server, follow the instructions in “Configuring 4D Remote as a Web Server” on page 39.
- 19 Depending on the network layer you are using (4D/NTK), there may be additional configuration to perform. See the relevant section below for instructions on what to do.
- 20 Restart 4D to activate your changes.

Updating an Existing Active4D 4.x/v5 Database

To update an existing Active4D 4.x/v5 database to Active4D v6, follow these steps:

- 1 Make a backup copy of your structure.
- 2 If you have made any changes to the Active4D shell, you will need to make note of the changes you made so they can be integrated into the new shell.
- 3 Follow the steps above for installing into a non-Active4D database.
- 4 If are upgrading from 4.x and you made changes in the old *A4D_DebugInitHook* method, note that the second parameter, *\$ioDialogTable*, is no longer used, and must be removed.
- 5 If you are using 4D’s web server to handle network communications, follow Step 1 of “Configuring for 4D’s Web Server” on page 37.

Installing the Predefined Session Handler

A custom session handler allows you to store sessions in the database. If you would like to do so, you may use the predefined session handler in the “SessionHandler” folder of the plugin distribution archive. You have a choice of using 4D methods or a library. Before using either one, you must create two tables:

a4d_session_id

Field name	Type	Attributes
id	Long Integer	Automatic index, Autoincrement
next	Long Integer	

a4d_sessions

Field name	Type	Attributes
id	Long Integer	Automatic index
data	BLOB	
expires	Real	Automatic index

After creating the supporting tables, to use 4D methods as the session handler, do the following:

- 1 Import the methods in the "SessionHandler/Methods" folder that begin with "sessionHandler". Name the methods according to the filename (without the extension).
- 2 Add the declarations in "SessionHandler/Methods/Compiler_Methods.txt" to your Compiler_Methods method (or whatever method you use for method compiler declarations).
- 3 Add the following line to your Active4D.ini configuration file:

```
session handler = sessionHandler
```

To use a library as the session handler, do the following:

- 1 Move "SessionHandler.a4l" to the "Active4D" folder of your database, or to one of the directories specified in the "lib dirs" configuration option in Active4D.ini.
- 2 Add the following line to your Active4D.ini configuration file:

```
session handler = *sessionHandler
```

For more information on session handlers, see "Session Handlers" on page 348.

Post-Installation Configuration

Active4D uses a special housekeeping process that takes care various internal tasks, including running special code when a web session expires. By default this process is given a stack size of 128K, which should be sufficient. If you need to change the stack size, do the following:

- 1 Open the List Editor.
- 2 Click on the list "A4D_Config".
- 3 Change the number in the item that says "HousekeeperStack=128" to the amount in kilobytes that you would like to reserve for the housekeeper's stack.
- 4 Close the List Editor and restart 4D.

Depending on the network layer you are using (4D/NTK), there may be some extra configuration necessary if you are installing Active4D for the first time.

Configuring for 4D's Web Server

If you are using the 4D web server to handle network communications, there are a few steps you must take:

- 1 Set the *On Web Connection* database method to the following code:

```
C_TEXT($1;$2;$3;$4;$5;$6)
A4D_OnWebConnection($1;$2;$3;$4;$5;$6)
```

- 2 Create a directory called "web_static" in the database folder if you are web serving with 4D Server or Standalone, or in the folder <shared 4D folder>/com.aparajita/Active4D if you are web serving with 4D Remote. The shared 4D folder is the directory within which the 4D licenses folder is found. You can use this directory for static content that you want to be served directly by 4D's web server.
- 3 Install the 4D Internet Commands plugin bundle in one of the plugins folders 4D searches for plugins.
- 4 Open your database and then open the Database Settings dialog. Consult the tables below and set each of the Web options according to the values in the tables.
- 5 Click OK to accept your changes, and then restart 4D.

Table 1: Configuration Preference Pane

Option	Value
Launch Web Server at Startup	[unchecked]
TCP Port	[as necessary]
IP Address	[as necessary]
Enable SSL	[as necessary]
HTTPS Port Number	443
Default HTML Root	web_static
Default Home Page	[empty]
Non-contextual Mode	[selected]

Table 2: Advanced/Options (I) Preference Pane

Option	Value
Use the 4D Web cache	[checked]
Pages Cache Size	[as desired]
Inactive Process Timeout	[as desired]
Web Passwords	No passwords

Table 3: Options/Options (II) Preference Pane

Option	Value
Send Extended Characters Directly	[checked]
Standard Set	[as desired, but must match Active4D character set settings]
Use Keep-Alive Connections	[as desired]
Number of requests by connection	[as desired]
Timeout (seconds)	[as desired]

Configuring for NTK

If you wish to use NTK as the network communications layer for Active4D, do the following:

- 1 Open your database and then open the 4D Preferences dialog.
- 2 Select the *Web/Configuration* preference pane.
- 3 Uncheck “Publish Database at Startup/Launch Web Server at Startup”.
- 4 Click OK to save the changes.
- 5 If you are using NTK 3.1+, open the project method “A4D_HTTPD_CompressResponse” and uncomment line 47.

NTK’s behavior as a web server can be configured using the 4D list “A4D_NTKConfig”, which contains the following items:

Item	Description
IP=	Defines the IP addresses on which NTK will listen. Leaving this empty will listen on any IP address. Specifying an address such as “192.168.1.13” will cause NTK to listen only on that address.
Port=8080	Defines the default port on which NTK will listen for HTTP requests.
SSLPort=443	Defines the default port on which NTK will listen for HTTPS (SSL) requests.
NumThreads=13	Number of HTTP listeners to preallocate.
MaxThreads=27	Maximum number of HTTP listeners to allocate.
ThreadStack=128	Stack size for HTTP listener processes in kilobytes.
CompressResponse=1	Determines whether responses are compressed with gzip. Change “1” to “0” to turn compression off.
KeepAliveTimeout=5	How many seconds a persistent (keep-alive) connection will remain idle before closing.

By changing the numbers in the various items, you can change how Active4D's shell sets up NTK.

If you are using NTK and would like to use SSL streams, you must follow these steps:

- 1 Open the method *A4D_InitSSLHook*.
- 2 Set *\$outCertPath*, *\$outCertKeyPath*, and *\$outCertPassword* to the values for your SSL certificate, SSL private key, and SSL password.
- 3 Close the method and restart 4D to have your changes take effect.

Configuring File Types to Compress

By default, the NTK shell gzip compresses all files that have a MIME type of "text/@", or whose MIME type appears in the 4D list "A4D_CompressContentTypes". If you wish to compress other file types, add the MIME types to the "A4D_CompressContentTypes" list and also be sure to add an entry in *ExtensionMap.ini* for each file type.

Configuring 4D Remote as a Web Server

Ordinarily Active4D becomes inactive on 4D Remote, since it assumes it is serving web pages from 4D Server. If you are using 4D Remote as a web server, you must explicitly tell Active4D that you are doing so.

To enable Active4D's web serving on 4D Remote, do the following:

- 1 Install the Active4D folder (containing the standard libraries and configuration files) onto the 4D Remote machine. The location of the Active4D folder on a 4D Remote machine must be in the standard search path as described in "Config File Search Path" on page 41.
- 2 Open the method *A4D_InitHook*.
- 3 Uncomment the line *\$ioClientsWebServer->:=True*. If no such line exists, add it to the method. Depending on your setup, you may need to set the value based on some set of conditions, as you may not want all 4D Remotes to be web servers.
- 4 Close the method.
- 5 If 4D passwords are active, open the 4D Preferences dialog. Select the Web/Advanced preference pane. When 4D passwords are active, 4D Remote will only give database access to the user selected in the *Generic Web User* drop-down list. It is up to you to select the proper user, then click OK to save your changes. It is also up to you to ensure that the 4D Remote acting as a web server will be logged in as the generic web user.
- 6 Restart 4D.

Using the Pre- and Post-Execute Hooks

For some developers it is necessary to examine and perhaps modify a request before and after Active4D. The Active4D shells provide hooks to allow the developer to do so without changing the core shell methods.

Pre-Execute Hook

The pre-execute hook allows you to inspect an incoming request. Within this hook you are given access to the raw request and are free to do whatever you want with it. If you decide to handle the request yourself, you can prevent Active4D from handling the request by returning *False* from the hook. If you return *True* Active4D will handle the request normally.

Note: If your aim is to inspect and possibly change the URL, a better way to do this is through the *On Request* event handler. For more information, see “On Request” on page 128.

Actually, there are two pre-execute hooks. The hook you will use depends on the network layer you are using.

If you are using 4D's web server as the network layer, you will use the hook method called *A4D_PreOnWebConnectionHook*. This method receives a *copy* of the requested URL, the request itself, and the remote address. Because the hook receives a copy of the request, any changes you make to your copy of the request will be ignored by Active4D.

Note: By default, the call to *A4D_PreOnWebConnectionHook* is commented out to prevent unnecessary copying of the request. If you plan to use this hook, you must uncomment its call in *On Web Connection*.

If you are using NTK as the network layer, you will use the hook method called *A4D_PreStreamExecuteHook*. This method receives a stream reference which you can use to read and modify the request. Since Active4D uses the same stream reference, any changes you make to the request will be used by Active4D.

Note: If you want Active4D to execute a request, you must be sure to put any data you read (or a modified version thereof) back into the stream so that Active4D will receive the entire request.

Post-Execute Hook

The *A4D_PostExecuteHook* method allows you to inspect and modify the response headers and response body returned by Active4D after it has executed, but before the response is sent. This method receives a pointer to the request and response header names and values (as parallel text arrays) and a pointer to the response body BLOB.

Within this hook you may make any changes you wish to the response headers or the response body, and those changes will be reflected in the response sent to the client.

If you change the response length, the Content-Length header must be changed accordingly. There is already code in the *A4D_PostExecuteHook* to do this, but it must be activated. If you do change the length of the response, do the following:

- 1 Locate the *If (False)* statement towards the bottom of the method.
- 2 Change *False* to *True*.

CHAPTER 3

Configuration

Active4D ships preconfigured to meet most needs, but you can configure virtually every aspect of Active4D to fit your specific needs.

Before discussing the various configuration (or “config”) files, you must know where to find them.

Config File Search Path

Active4D looks for each config file using the following search path:

- <Database folder>/Active4D (Standalone/Server)
- <User home>/Library/Application Support/4D/com.aparajita/Active4D/conf (macOS)
- <Disk>:\Users\<user>\AppData\Roaming\4D\com.aparajita\Active4D\conf (Windows 7 or Vista)
- <Disk>:\Documents and Settings\<user>\Application Data\4D\com.aparajita\conf (Windows XP)
- /Library/Application Support/4D/com.aparajita/Active4D/conf (macOS)
- <Disk>:\ProgramData\4D\com.aparajita\Active4D\conf (Windows 7 or Vista)
- <Disk>:\Documents and Settings\All Users\Application Data\4D\com.aparajita\Active4D\conf (Windows XP)
- Active4D.bundle/Contents/Resources/conf

Note: Active4D will follow aliases and symbol links on macOS and shortcuts on Windows.

This arrangement allows you to have multiple versions of config files, with those earlier in the search path overriding those later in the search path.

So, for example, you could establish a baseline set of configuration options which you keep in the shared 4D folder, then override that configuration for a particular database by putting a separate config file in `<Database folder>/Active4D`.

Warning: If 4D Remote is being used as a web server, you must install the config files on the 4D Remote machine.

Configuration Files

Active4D has four config files: *Active4D.ini*, *ExtensionMap.ini*, *Realms.ini* and *VirtualHosts.ini*. These are plain text files which may be created and edited on any platform, as Active4D will accept Mac, Windows and Unix line endings.

Default versions of these files which should suit most needs are shipped with Active4D. Each file is heavily commented, so if this documentation is not handy you can always read the comments. It is recommended that you keep a copy of the default config files somewhere safe in case you need to go back to a baseline configuration.

Active4D periodically checks to see if the config files have been modified. The interval between checks is set with the “refresh interval” option in *Active4D.ini*. If a config file has been modified, it is reloaded automatically, so there is no need to quit and restart 4D to have the changes take effect.

Note: The config files are loaded on startup and their locations are cached, so you may not move them without restarting the server.

The Default Directory

Some of the config files rely on the concept of the *default directory*. The default directory is the directory from which relative paths are resolved.

Under 4D Standalone or Server, the default directory is the current database directory (the one with the structure file). Under 4D Remote, the default directory defaults to:

- `/Library/Application Support/4D/com.aparajita/Active4D` (macOS)
- `<System Disk>:\ProgramData\4D\com.aparajita\Active4D` (Windows 7 or Vista)
- `<System Disk>:\Documents and Settings\All Users\Application Data\4D\com.aparajita\Active4D` (Windows XP)

If 4D does not have sufficient permissions to create the Licenses folder at the above path, or if you are running under 4D Volume Desktop Client and there is no Licenses folder in the merged application, the default directory is:

- `<User home>/Library/Application Support/4D/com.aparajita/Active4D` (macOS)
- `<System Disk>:\Users\Current user\AppData\Roaming\4D` (Windows 7 or Vista)
- `<System Disk>:\Documents and Settings\Current user\Application Data\4D`

To find out which path is used for the default directory, look in the Active4D log for a line like this:

```
Sep 22 15:33:13 Active4D: [info] interpreter: default directory:
/Users/web/Library/Application Support/4D/com.aparajita/Active4D
```

Path Format

All paths specified in config files must be in URL format (also known as Posix format), which has the following attributes:

- The directory separator is '/'
- A full path begins with a leading '/', a relative path does not
- A directory path may or may not be terminated with a trailing '/'
- A file path must not be terminated with a trailing '/'
- The path may not contain the native directory separators ':' and '\'
- The path may begin with a special directory token, discussed below
- The path may contain directory movement (../)
- Any element of a path may be an alias or symbolic link in macOS, or a shortcut in Windows

There are four tokens you may use at the beginning of a path to represent special directories. The tokens are:

Token	Directory
<default>	The default directory, as defined in "The Default Directory" on page 42
<web>	The web root directory
<4d volume>	The name of the volume on which the 4D application resides
<boot volume>	The name of the system boot volume

The tokens include the '<' and '>', and are replaced with the directories they represent, without a trailing '/'. This allows you to create paths that are portable across machines and 4D directories.

For example, you may want to include files from a folder called "includes" outside of the web root, perhaps at the same level as the web root folder. To do so you would specify this path:

```
<web>/../includes
```

To help you understand how URL paths map to the Mac or Windows paths used by 4D, here are a few samples of full paths in 4D format and URL format, where "Dev" is the name of the boot volume on macOS:

Native Full Path	Platform	URL Path
Dev:site:web:	Mac	/site/web
Dev:site:web:index.a4d	Mac	/site/web/index.a4d
Images:02:0213.jpg	Mac	/Volumes/Images/02/0213.jpg
C:\Dev\site\web\	Win	/C/Dev/site/web
C:\Dev\site\web\index.a4d	Win	/C/Dev/site/web/index.a4d
D:\02\0213.jpg	Win	/D/02/0213.jpg

Native Relative Path	Platform	URL Path
:site:web	Mac	site/web
site\web\	Win	site/web

Note: On macOS, full paths on the boot volume may begin either with `"/Volumes/<volume name>/"` or just `"/"`, whereas full paths on non-boot volumes *must* begin with `"/Volumes/<volume name>/"`. The first time Active4D converts a Posix path to a 4D path, the boot volume name is looked up and then cached. If you wish to rename your boot volume while 4D is running, you must restart 4D to ensure path conversions are correct.

The Standard Search Path and Path Lists

Several config options allow you to specify a semicolon-delimited list of full paths to search in addition to the standard search path. If such a list is provided, it is searched after `<Database directory>/Active4D`.

For example, to add two directories to the search path, you might use something like this:

```
<default>/libs;<boot>/Library/Application Support/4D/libs
```

Active4D.ini

The config file `Active4D.ini` controls most of the behavior of Active4D and must be in an Active4D folder in the standard search path. There are several dozen options you can set in this file. Each option is in the form:

```
<option> = <value>
```

Case is not significant either for keys or values, and you can have any amount of white space surrounding the `"="`. You may also use line comments and block comments just as you would in 4D and Active4D.

The options in Active4D.ini are listed below. A detailed discussion of the options can be found in the Active4D.ini file itself.

Table 1: Error Handling Options

Option	Description
error page	The root-relative URL path to a file to execute when an error occurs
http error page	The root-relative URL path to a file to execute when an HTTP error occurs
log level	A sum of the bit flags defining which type of events to log

Table 2: HTTP Server Options

Option	Description
allowed methods	A list of allowed HTTP request methods
cache control	The default cache control for executable files
default page	Name of default page when directory URL is requested
doctype	Indicates the doctype of the site ("html" or "xhtml")
executable extensions	File extensions which Active4D will consider executable
expires	The number of minutes before a page should expire
fusebox page	Page through which Fusebox requests go
max request size	The maximum size in kilobytes (K) of a request, including the headers
parameter mode	Which collections to use for form variables and query string params
parse json request	Whether to automatically parse JSON requests
receive timeout	Maximum wait time when receiving from a TCP stream
root	Root web directory
serve nonexecutables	Whether Active4D should serve non-script files

Table 3: Interpreter Options

Option	Description
auto relate one	Whether to load related records in auto-related one tables
auto relate many	Whether to load related records in auto-related many tables
output charset	The character set to convert to when writing text to the response buffer
output encoding	The encoding to perform on text written to the response buffer

Table 3: Interpreter Options

Option	Description
platform charset	The character set from which source files are converted to Unicode
script timeout	The minimum script timeout in seconds

Table 4: Library Options

Option	Description
auto refresh libs	Whether to automatically refresh libraries
lib dirs	Extra search paths for libraries
lib extension	The extension for Active4D libraries
refresh interval	How often to check libraries and config files for modification

Table 5: Security Options

Option	Description
auto create vars	Whether to create local variables from form variables and query params
safe doc dirs	Directories allowed for document commands outside of root
safe script dirs	Script directories outside of root

Table 6: Session Options

Option	Description
session cookie domain	The domain of the session cookie
session cookie name	The name of the cookie to store with the session ID
session cookie path	The path of the session cookie
session cookie secure	If true, the session cookie is marked secure and is only sent in secure requests
session handler	The base name of 4D methods or the name of an Active4D library if prefixed with "*"
session purge interval	The minimum interval in seconds between attempts to purge expired sessions
session timeout	The length of time in minutes that a session can live without any user interaction

Table 6: Session Options

Option	Description
session var name	The name of the local variable to automatically set to the current session ID
use session cookies	Whether Active4D should store session IDs in cookies
use sessions	A global switch for session management

ExtensionMap.ini

This config file maps filename extensions to MIME types. It is used primarily by Active4D's HTTP server. For more information on this file, see "ExtensionMap.ini" on page 59.

Realms.ini

This config file specifies the mapping between realm names and portions of a URL that identify those realms. It is used primarily by Active4D's HTTP server. For more information on this file, see "User Authentication" on page 59.

VirtualHosts.ini

This config file provides a routing table for virtual hosting, which allows you to create multiple independent web sites on the same machine. For more information on this file, see "VirtualHosts.ini" on page 61.

CHAPTER 4

Security

Security is something you should worry about with any web site. When a web site has direct access to a mission-critical database, security is a mission-critical issue. Active4D provides several mechanisms to ensure your site is as secure as possible.

Source Code Security

When you build a web site with Active4D, all or part of your source code is on the web server in plain text form.

This does not mean your source code is not secure.

In determining the potential security risk, you must keep in mind these two important facts:

- A web site is only as secure as its web server.
- Active4D removes the source code before serving the page.

Web Server Security = Source Code Security

If an attacker compromises the security of your web server, your source code is at risk — but then, so is everything else on the web server, including the database.

If the attacker is bent on being destructive, there are far easier and more effective ways to wreak havoc than to change the embedded source code in a web page. And it is highly unlikely that the hackers of the world know the 4D language.

Circumventing Active4D

Active4D always strips out source code before sending a page to the client. But what if the page is served directly without passing through Active4D?

It is up to you to ensure that every request is passed to Active4D.

If you are using 4D's built-in web server as the network layer, there is a potential risk. Active4D is only invoked through the *On Web Connection* mechanism. *On Web Connection* is only invoked when the user requests a URL that does not exist. URLs that reference existing files are served directly by 4D without invoking *On Web Connection*.

Warning: If 4D's web server has its default HTML root set either to nothing or to your real web root, there is nothing to prevent a casual user from reading your source code.

For example, let's suppose the user is given this URL:

```
www.myserver.com/4DCGI/mypages/mypage.a4d
```

Let us further suppose that this page exists within a directory called "web" within the database directory, and the 4D web server's default HTML root is set in the Database Properties dialog to "web". If the user then enters the same URL without "/4DCGI", like this:

```
www.myserver.com/mypages/mypage.a4d
```

4D will directly serve that page, complete with embedded source code, without going through *On Web Connection*. On the other hand, if 4D's default HTML root is set to nothing, the user could still access the source with this URL:

```
www.myserver.com/web/mypages/mypage.a4d
```

It is critical that you create an empty "decoy" directory in your database directory and set 4D's default HTML root to that folder, then set Active4D's root directory to something else. By doing this, you ensure that all URLs without "/4DCGI" will be to non-existent files, thus forcing 4D to route the request through *On Web Connection* and ultimately to Active4D.

Potential Attacks

Even if a miscreant does not succeed in hacking into your web server, there are still many opportunities for mischief. For an excellent essay on the security issues arising from a web-based scripting language, I recommend reading the following:

<http://www.secureality.com.au/studyinscarlet.txt>

If you do read this essay, please note that Active4D addresses almost every potential attack outlined there.

Executing/Accessing Non-Web Files

One potential source of attack is to allow access to files outside of the web site itself. Ordinarily, Active4D requires that any requested file be within the web root directory or a subdirectory thereof. This includes files that are executed with the **include** or **require** commands.

If an attempt is made to execute a file in an unauthorized directory, Active4D aborts execution and returns the status code *403 Forbidden* along with a message indicating the error.

The "safe script dirs" Config Option

If you want to execute or include script files in a directory outside of the root directory, you can provide a list of allowable script directories with the "safe script dirs" option in Active4D.ini.

If an attempt is made to execute a file outside of the root directory, this path list is checked. If the file lies within one of the specified directories, the execution is allowed to proceed. Otherwise the behavior is as specified above for unauthorized directories.

By default the “safe script dirs” path list is empty.

Misusing Document Commands

Let us suppose that you have a script that deletes a file, and that the path to the file is kept in the session. An attacker could potentially force the session to contain a path to a critical system file outside of the web root directory.

To prevent this kind of attack, the path passed to all document commands (, **DELETE DOCUMENT**, etc.) is checked to ensure that the path is within the web root directory or a subdirectory thereof.

If an attempt is made to use a document command on an unauthorized directory, Active4D does the following:

- Aborts execution of that command, but continues executing the script
- Sets *OK* to zero
- Sets the variable *A4D_Error* to -45 (File is locked)

As a result of this behavior, you should at least check the *OK* variable when using any document commands.

The “safe doc dirs” Config Option

If you want to use document commands on files outside of the root directory, you must add the directory path to the “safe doc dirs” path list in Active4D.ini. For information on Active4D.ini, see Chapter 3, “Configuration.”

If a document command is given a path outside the root directory, this path list is checked. If the path lies within one of the specified directories, the command is allowed to proceed. Otherwise it behaves as specified above for unauthorized directories.

By default the “safe doc dirs” path list is empty.

Spoofing Form Variables

Suppose you have a form which posts some critical piece of information in a hidden form variable called “f_id”. In the form processing script, if the “auto create vars” option is on, you would access the local variable called *\$f_id* to get the hidden value, then use this ID to perform some critical operation.

An attacker could execute the form processing script, passing in a query string with a parameter “f_id” and some bogus value. Because Active4D ordinarily creates local variables from both query parameters and form variables, your script would have no way of knowing that the local variable *\$f_id* did not come from a hidden form variable. Depending on what you do with *\$f_id*, this could present a potential security breach.

The “auto create vars” Config Option

To prevent variable spoofing attacks, by default the “auto create vars” option in Active4D.ini is turned off. If you have specifically turned this option on, it is recommended that you turn it off and directly access form variables through the **_form** collection. For more information, see “Form Variables” on page 201.

Uploading Huge Files

If you have a web page which provides file upload, an attacker could attempt to upload a huge file. Most web servers buffer the request, thus it is possible to compromise the stability of the web server by forcing it to run out of memory.

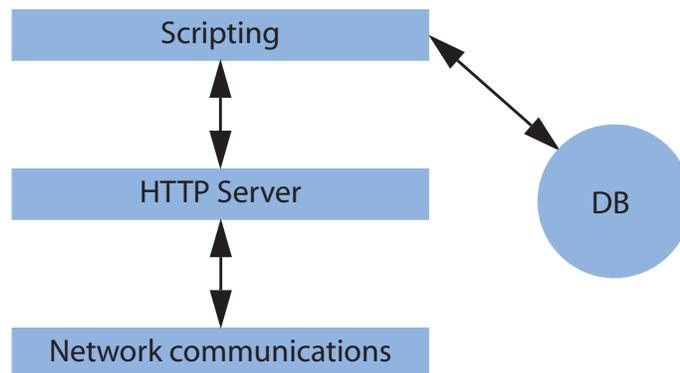
Active4D provides an option called “max request size” in Active4D.ini, which limits the total size of the request Active4D will accept. By setting this option at something reasonable for your particular application (the default is 64KB), you can prevent the attack outlined above.

If a request is received which is larger than *<max request size>* bytes, Active4D aborts execution and returns the status code *413 Request Entity Too Large*, along with an error message indicating that fact.

CHAPTER 5

HTTP Server

You may remember from the introduction the diagram of the various layers that make up a web scripting system. That diagram is reproduced here.



Both the HTTP server layer and scripting layer are handled fully by Active4D.

What Is a Web Server?

You may be wondering how Active4D differs from 4D's built-in web server or from dedicated web servers like apache.

Traditionally a web server handles the following tasks:

- Network communications between the host and the client
- Parsing of HTTP requests and obeying the rules of the HTTP protocol based on the contents of those requests
- Finding the resource requested and returning it in a response with the appropriate HTTP response headers
- Providing a means of specifying realms of the web site that require user authentication
- Error handling
- Virtual host routing

Active4D + Network Layer = Web Server

Active4D relies on the host to provide network communications. Other than that, Active4D handles all of the task listed above, right out of the box, with no programming

required. And because Active4D is the web server, you have full access to every aspect of both the HTTP request and the HTTP response from within Active4D.

Believe me, implementing a web server with all of those features using other tools is not a trivial exercise!

The Active4D Shell comes preconfigured to use 4D's web server. If you already are using NTK and wish to use it with Active4D, it can be used as the network layer as well.

HTTP Fundamentals

To understand the functioning of a web server, it really pays to understand how HTTP requests work. David Adams has been kind enough to allow me to include Chapter 47 of his most excellent book, *The 4D Web Companion*. Before going on, I recommend taking the time to read through this excellent introduction to HTTP, which is included in the Active4D documentation distribution as a separate PDF document, "HTTP Fundamentals.pdf."

Note: You don't need to pay attention to his coverage of 4D web commands and tags, as you will have no need for them with Active4D.

Active4D HTTP Request Handling

Now that you know a little bit about HTTP requests, we can look at how Active4D handles an HTTP request. Before a request can be handled, a TCP/IP connection must have established between the *host* and a *remote* client.

Note that the descriptions below will refer to *collections*, which are special data structures used throughout Active4D. It is enough for now to know that a collection is an associative array of key and values. For more information on collections, see "Collections" on page 99.

At any step of the request handling, if Active4D detects either corrupt data or data that does not conform to the HTTP specification, the status *400 (Bad Request)* is returned immediately.

Executable vs. Non-executable Files

When Active4D is asked to serve a file, it first checks the file's extension against the list of executable extensions specified in Active4D.ini.

There are several reasons for determining if a file is considered executable or not:

- **Efficiency:** It is inefficient to initialize and run the Active4D interpreter for a file that has no executable code.
- **Stability:** If Active4D is asked to parse a file that is not textual, such as a GIF file, it is possible that it could be fooled into executing nonsense, which would lead to an error condition.

- **Convenience:** Active4D can be configured to serve non-executable files (including binary files) and does all the work that an HTTP server should do, such as generating proper headers, so you don't have to worry about it.

Note: If you are using 4D's web server with Active4D, it is more efficient to put non-executable files in the `web_static` directory so that 4D can serve them directly.

Request Header Parsing

The first step in handling a request is to retrieve and parse the request header.

- 1 The network layer invokes Active4D, passing in some basic context information like the host and remote IP address of the current connection.
- 2 Two response headers are created: "X-VERSION: HTTP/1.1" and "X-STATUS: 200 OK". The response status is initialized to 200 (OK).
- 3 If Active4D has been asked to retrieve the request via a TCP/IP stream, it retrieves the request headers using a receive callback method. Otherwise it is given the entire request. If the request size is greater than the size set in the "max request size" option in Active4D.ini, a status of 413 (Request Entity Too Large) is returned immediately.
- 4 The request method is parsed. If the method is missing or malformed, a status of 400 (Bad Request) is returned. Otherwise the request method is saved in the **request info** collection.
- 5 If the request method is not supported by Active4D at all, a status of 501 (Not Implemented) is returned. Otherwise it is checked against the list of allowed methods as specified by the "allowed methods" option in Active4D.ini. If the request method is not in the allowed methods list, a status of 405 (Method Not Allowed) is returned.
- 6 The requested URL and HTTP version are parsed and saved.
- 7 The request headers are parsed. Each header and its associated value is put into the "request info" collection.
- 8 If a "Cookie" header is present, the cookies are parsed out and their names and values are put into the "request cookies" collection.
- 9 If an "Authentication" header is present and the authentication type is "Basic", the authentication username and password are decoded and saved.
- 10 If an "X-Requested-With" header is present with the value "XMLHttpRequest", the "*ajax" item in the *request info* collection is set to true, the content charset is set to "UTF-8", and the output charset is set to *A4D Charset UTF8*.

Note: Every major browser and Javascript toolkit tested automatically sets this header, you should never have to worry about setting it yourself.

- 11 If the request method is "POST" or "PUT" and the "Content-Length" header indicates a request size larger than the "max request size" option in Active4D.ini, a status of 413 (Request Entity Too Large) is returned immediately. If you are using **A4D**

Execute Stream Request, the receive callback is called to read the rest of the response before returning the 413 response.

- 12 If the request is a CORS request (it has an “Origin” header), the full URL including protocol and host is assembled and matched against the entries in Cors.ini. If any entries match, the appropriate CORS response headers are set.
- 13 If the URL contains a query string, the query string parameters are parsed and put either into the *query params* or *form variables* collection, depending on the “parameter mode” option in Active4D.ini.
- 14 If an On Request event handler is defined it is called, and if a status < 200 or > 206 is returned, that status is returned immediately. For more information on the *On Request* event handler, see “On Request” on page 128.
- 15 If the filename of the requested URL has an extension and the request method is not “OPTIONS”, the extension is checked against the “executable extensions” option in Active4D.ini. If it matches an extension there, it is considered executable. Otherwise it is considered non-executable.

If the file is considered executable, what occurs next depends on the network layer used in conjunction with Active4D.

POST/PUT and File Upload Handling

When Active4D is used in conjunction with 4D’s web server, a callback method is used to call **GET HTTP BODY**, which retrieves the request body. When using NTK, the shell passes the NTK stream reference to Active4D, which it then uses to retrieve the request data as needed via a callback method.

- 1 Active4D uses a receive callback method to receive the entire request body.
- 2 Active4D determines what type of post was made, based on the Content-Type header: regular form (www-form-urlencoded), multipart form (multipart/form-data), JSON (application/json), or raw upload (all other content types).
- 3 If the post is a regular or multipart form, the form variable data in the request body is parsed, and the names and values are put either into the *form variables* or *query params* collection, depending on the “parameter mode” option in Active4D.ini. If the request is an Ajax request, the form variables are automatically converted from UTF-8.
- 4 If a multipart form contains uploaded files, they are saved temporarily to disk.
- 5 If the post is JSON and the “parse json request” option is true, the posted data is parsed (in the same way as the **parse json** command) and the result is placed into the *form variables* or *query params* collection (depending on the “parameter mode” option in Active4D.ini) under the name “_json_”. If the “parse json request” configuration option is false, the post is treated as raw data.
- 6 If any other type of post was made, the raw data in the post is placed into the *form variables* or *query params* collection (depending on the “parameter mode” option in Active4D.ini) as a BLOB under the name “_data_”.

Executable Request Handling

If an executable request has been received and the header and body successfully parsed, Active4D proceeds to the next step.

- 1 Context information such as host and remote IP address is put into the “request info” collection.
- 2 The cache-control, expires, and content charset settings are initialized from the configured values in Active4D.ini. The content type is set to “text/html”.
- 3 If virtual hosts have been defined, Active4D sets the root directory, virtual host name and default page from the virtual host routing table specified in VirtualHosts.ini. For more information on virtual hosts, see “Virtual Hosting” on page 61.
- 4 If the URL is prefixed with “/4DCGI”, the prefix is removed from the URL.
- 5 If the remaining URL is absolute (begins with ‘/’), the current root directory path is inserted at the beginning of the URL. This forms a full path to the requested resource.
- 6 If the URL ends with ‘/’, it is assumed to be a directory. If the URL was mapped by VirtualHost.ini, the default page specified there is appended to the directory. If the URL was not mapped, the next default page in the list of default pages is appended.
- 7 The full path is converted from URL to native format and any aliases in the path are resolved to their targets. The resolved target path replaces the unresolved path.
- 8 The full path is checked to make sure it is either in the root directory or in one of the directories specified in the “safe script dirs” option in Active4D.ini. If it is not, a status of 403 (Forbidden) is returned immediately.
- 9 The full path is checked to see if it is a valid directory path. If so, a trailing ‘/’ is added and a redirect is issued to that URL.
- 10 If the full path is invalid and the request was not mapped by VirtualHosts.ini and there are more default page names to try, the server goes back to step 6. Otherwise 404 (Not Found) is returned immediately.
- 11 The current realm (if any) is set from the realm map in Realms.ini. For more information on realms, see “User Authentication” on page 59.
- 12 If the full path is valid, the file is opened and the interpreter is invoked to execute the file. For information on what happens during execution, see Chapter 7, “Interpreter.”
- 13 If a graphic has been written to the response buffer and the response buffer is empty, something went wrong during the graphic conversion process and the response status is set to 404 (Not Found).
- 14 If the status is ≥ 400 (an error), skip to Step 21.
- 15 If cache-control has been set, a “Cache-Control” header is added to the response headers. Since older browsers do not obey the “Cache-Control” header, if the cache-control setting is “no-cache”, a “Pragma” response header with the value “no-cache” is added, and the expires time is set to now.
- 16 If a content type has been set, a “Content-Type” response header is added.
- 17 If a charset has been set, it is appended to the “Content-Type” response header.
- 18 If an expires date has been set, an “Expires” response header is added.
- 19 The headers in the “response header” collection are added.

- 20 The cookies in the “response cookies” collection are added as “Set-Cookie” response headers.
- 21 The “X-STATUS” response header is set to the response status.
- 22 If the request method is “HEAD”, the response buffer is cleared. In this case, or if the response status is less than 400 (no error), skip to Step 25.
- 23 If the status is ≥ 400 (an error), Active4D attempts to find a file in the root directory with the name `<status>.html`, where `<status>` is the numeric status. Thus with a 404 (Not Found) status, Active4D would look for a file called “404.html”.
- 24 If such a file is found, it is placed in the response buffer, and a “Last-Modified” response header is added. If such a file is not found and the HTTP spec says that an error message should be returned for the current status, an appropriate error message is placed in the response buffer.
- 25 A “Content-Length” response header is added which contains the size of the response buffer.
- 26 A “Content-Type” response header is added with the MIME type of the response.
- 27 Control is returned to 4D. The shell code sends the response headers and then the contents of the response buffer. If 4D 2003 is the network layer and the “Content-Type” response header is “text/html”, 4D tags are stripped from the response.

Non-Executable Request Handling

By default, Active4D will serve static content if given a path to a non-executable file. If you would like to serve static content yourself, you can do so by changing the “serve nonexecutables” option to “false” or “no” in Active4D.ini, in which case Active4D does not serve the file, but provides you with the full native path and modification date/time of the requested file.

When it is determined that a request for a non-executable (static) file has been made, Active4D does the following.

- 1 If the requested file is not found, a status of 404 (Not Found) is returned immediately.
- 2 If the requested file is found and the “serve nonexecutables” option is “false” or “no”, Active4D returns the full native path of the file, the file’s modification date and the modification time. For more information, see “outHeaderNames/outHeaderValues” on page 67.
- 3 If the requested file is found and the “serve nonexecutables” option is “true” or “yes”, the file’s extension is checked. If it has an extension and the extension was specified in ExtensionMap.ini, the content type of the file is set accordingly.
- 4 If no content type could be determined, the status is set to 404 (Not Found) and the file will not be served for security reasons.
- 5 A “Content-Length” response header is added which contains the size of the response buffer.
- 6 A “Content-Type” response header is added with the MIME type of the response, as specified in ExtensionMap.ini.
- 7 If the “nonexecutable expires” option is empty or is greater than zero, a “Cache-Control” header is added with “max-age” set to the value of the “nonexecutable expires” option.

- Control is returned to 4D. The shell code sends the response headers and then the contents of the response buffer.

Configuration

The Active4D web server is configured through the several config files: Active4D.ini, Cors.ini, ExtensionMap.ini, Realms.ini and VirtualHosts.ini.

Active4D.ini

There are several config options in Active4D.ini that determine the basic behavior of the web server. For information on the format of config options, see Chapter 3, “Configuration.” For a detailed description of each option, see the comments in the Active4D.ini file itself.

Cors.ini

Cross-Origin Resource Sharing (CORS) is a standard developed to allow servers to allow cross-domain requests via XMLHttpRequests (Ajax). Ordinarily XMLHttpRequest will not allow a request to another domain. CORS allows this if the server allows it.

A good introduction for CORS can be found here:

https://developer.mozilla.org/En/HTTP_access_control

Active4D implements full CORS support in its HTTP server. Details on how to configure CORS support can be found in the Cors.ini file.

ExtensionMap.ini

When a web server sends a file to the client browser, it must inform the browser of the file's type. Similarly, when a client uploads a file to the web server, the web server receives the file's type. This type is sent and received as a *MIME type*. MIME types are standardized and can identify a file's type independently of its filename extension.

ExtensionMap.ini is a config file that allows Active4D's web server to map filename extensions to MIME types. For example, let us say that Active4D receives a request for the file “logo.png”. It looks into its extension map and finds an entry which says that the corresponding MIME type is “image/png”.

Details on the format of the extension map entries can be found in the ExtensionMap.ini file.

User Authentication

Active4D provides full support for protecting portions of your web site against intruders by using the basic HTTP authentication protocol. Currently authentication only works with executable files.

For example, you may have an admin section of your web site that only administrators should have access to. Another section of your web site is only for accounting users. By

defining *realms* — sections of the web site defined by a hostname or substring of a requested file's path — you can automatically be notified when a user's credentials need to be authenticated.

Note: The basic HTTP authentication protocol has a number of security problems if it is not used over a secure SSL connection. In addition, each browser handles user authentication differently. Therefore it is recommended that you use your own form-based authentication if you need maximum control over the process.

The user authentication mechanism requires three elements that work together:

- Entries in `Realms.ini`
- `On Authenticate` event handler
- Use of the **authenticate** command

For information on the `On Authenticate` event handler, see “On Authenticate” on page 129. For information on the **authenticate** command, see “User Authentication” on page 410.

Realms.ini

This config file specifies the mapping between realm names and substrings of a path that identify those realms.

Each entry in `Realms.ini` has two fields in this format:

Realm<tab>*Match string*

The realm is a logical name for the section of the web site you wish to protect. It need not have any relation to an actual directory name. Realm names should not have any extended characters or spaces. If there are extended characters they are ignored. Spaces are converted to underscores.

You use the realm name internally to determine what section of the web site is being accessed. The browser caches user credentials for each realm, so subsequent authorized accesses in the same realm do not result in repeated requests for credentials.

If the match string begins with “?”, it is matched against the query string portion of the requested URL. If any portion of the query string matches, the corresponding realm is set.

If the match string does not begin with “?”, it is matched against the host name and the fully resolved path of the requested URL. If any portion of the host name or path matches the match string, the corresponding realm is made the current realm.

Matching is case-insensitive, but the entire match string must be found within the query string, host name or path.

Note: The realm is matched to the path which results after virtual host mapping, after the default page is added to a directory request and after any aliases have been resolved.

Because the match string could be anywhere in the path, you must ensure that your web directory names and match strings are selected such that the match strings do not inadvertently match the wrong directory.

In addition, because the realm entries are scanned in order, realms cannot be nested. For example, you may not put the directory *admin* in the “admin” realm and the subdirectory *admin/accounting* in the “accounting” realm. Only one of them will ever be selected.

If a realm is matched and the *On Authenticate* event handler has been defined, it is called before execution begins to allow you to authenticate the user.

If the match string is not found, there is no current realm and the authentication mechanism is not invoked.

Let’s look at some example *Realm.ini* entries.

Realm	Match string
secure-server	www.myserver.com
admin	/admin/

The first entry would place the entire host *www.myserver.com* in the realm “www.myserver.com”. That means any request made on that server would have to be authenticated for that realm.

The second entry places the directory *admin* in the “admin” realm. Note the use of the leading and trailing slashes to ensure only a directory is matched in the path. Don’t worry if the user enters *www.myserver.com/admin*, because Active4D redirects that to *www.myserver.com/admin/* which will match correctly.

Virtual Hosting

If you create a large database in 4D, most likely it has more than one module. Often these modules are restricted to a certain set of users.

When bringing a database to the web, you will want to carry the same structure to your web site. You could accomplish this by directing the different classes of users to separate subdirectories within the root directory. But wouldn’t it be nice to have completely separate web sites for each one?

Virtual hosts allow you to host more than one web site on a single machine and a single instance of 4D. Each virtual host can be mapped to a different web root and/or different default page.

VirtualHosts.ini

This config file specifies a routing table which determines where Active4D should route HTTP requests. The criteria is used to determine the routing are:

- **hostname:** This would be used if the client enters a domain name in the URL, such as *www.active4d.net*.
- **IP address:** This would be used if the client enters a direct IP address in the URL, such as *192.168.1.7*.

- **language:** This would be used if you have partitioned a single domain/IP address into separate sites based on the preferred language the client has specified in the browser. The language is specified as an ISO language code. A complete list of codes can be found in “ISO Language Codes” on page 629.

Since this is a routing table, you must specify the target of the routing. The complete format for a virtual host entry consists of a line with five tab-delimited fields:

```
IP Address[:port]<tab>Hostname[:port]<tab>Language<tab>Root|Host<tab>Default
```

There may be any number of tabs between fields. Line and block comments may be used between entries as well.

The target of a given routing is specified by two fields:

- **Root|Host:** The *Root|Host* field may contain two values separated by “|” (vertical bar).

If this field does not contain “|”, the entire contents of the field is used to specify the web root. If the field contains “|”, the portion to the left of the “|” specifies the web root, and the portion to the right of the “|” specifies the virtual host name.

The web root portion should be a URL-style (Unix) path. If it is relative (not beginning with “/”) it is relative to the default directory (see “The Default Directory” on page 42). You may use any of the path tokens (such as <4d volume>) that are valid for the root option in Active4D.ini, or you may use “*” (without quotes) to use the value of the “root” option in Active4D.ini.

The virtual host name portion can be any text (including spaces). The specified name is available through the “*virtual host” item in the *request info* collection. This is useful, for example, if you are using common templates for each virtual host but skinning them individually. You can use the virtual host name as a means of building a path to images and stylesheets for each virtual host.

- **Default:** This field specifies the filename of the page that should be used as the default page (when a directory is requested) for the given virtual host. Using “*” (without quotes) will use the value of the “default page” option in Active4D.ini.

You may use a value of * in the *IP Address*, *Hostname* and *Language* fields to ignore that field when matching.

The entries are searched in order, so they should progress from most specific to the least specific. Hostname matching is case-insensitive and Language matching is case-sensitive. Both the *IP Address* and *Hostname* fields accept standard 4D wildcard characters to allow partial matches. Otherwise the entire field must match. An invalid or missing value will result in that entry being ignored.

The *Hostname* field must include the port if the host is on a port other than 80.

The final entry in the table should be terminated with a line ending.

If a match is made with the *IP Address*, *Hostname*, and *Language* fields, the *Root|Host* and *Default* fields are used to set the web root, virtual host, and default page.

A Virtual Host Example

To understand how virtual host routing works, let's look at some examples for a site that has an English and French version. We are using multihoming to have multiple IP addresses on the same machine. In addition, we will define three hosts, *mac.home.com*, *mac.admin.com* and *mac.home.fr*. The first two are on IP address 192.168.1.7, and the third is on 192.168.1.27.

//IP address	Hostname	Lang.	Root	Default
*	*	fr	web_fr fr	default.a4d
*	mac.home.com	*	* main	*
*	mac.admin.com	*	admin admin	*
192.168.1.7	*	*	* main	*
*	mac.home.fr	*	web_fr fr	default.a4d
192.168.1.27	*	*	web_fr fr	default.a4d
*	@.foo.com	*	web_foo	*
192.168.2.@	*	*	web_foo	*

The first entry routes all requests where the user's browser is configured with French as the preferred language. The directory "web_fr" is made the root directory for those requests, the virtual host name is set to "fr", and a URL to a directory will redirect to the default file "default.a4d" in that directory.

The second entry will route all requests to *mac.home.com*. The root directory and default file will be set to whatever they are configured to be in Active4D.ini. The virtual host name will be "main".

The third entry will route all requests to the virtual host *mac.admin.com*. The root directory will be set to "admin", the virtual host name will be "admin", and the default file will be set to whatever it is configured to be in Active4D.ini.

The fourth entry will route all other requests to the IP address 192.168.1.7 in the same way the second entry does. For completeness, you should always have an entry for both the hostname and IP address to catch all possible requests.

The fifth and sixth entries route all requests to the host *mac.home.fr* or the IP address 192.168.1.27. The directory "web_fr" is made the root directory for those requests, the virtual host name will be "fr", and a URL to a directory will redirect to the default file "default.a4d" in that directory.

The final entries route requests to the domain "foo.com" or to the subnet 192.168.2 to the web root "web_foo". The virtual host name will be empty.

HTTP Error Handling

If an http error occurs during the processing of a request, before any scripts are executed, Active4D returns the status code indicating the error that occurred and a default error message where indicated by the HTTP specification. In addition, Active4D adds two response headers:

- **X-Error-Status:** This header contains the HTTP error status code.
- **X-Error-URL:** This header contains the full requested URL that caused the error.

Customizing the Error Handling

You can override the default HTTP error message in several ways:

- **http error page:** If you set the “http error page” option in Active4D.ini, Active4D will attempt to execute that page if it is executable or load it statically if not. If you have not specified anything in the “http error page” option, or you have and the page cannot be found or an error occurs during its execution, Active4D moves on to the next option.
- **<status>.<ext>:** The next option is to attempt to find a file called <status>.<ext> in the current web root directory, where <status> is the HTTP status code (such as 404) and <ext> is the first executable extension configured with the “executable extensions” option in Active4D.ini, or “.a4d” if none have been configured. If no such file exists, or if it exists and an error occurs during its execution, Active4D moves on to the next option.
- **<status>.html:** The next option is to attempt to find a static HTML file called <status>.html in the current web root directory, where <status> is the HTTP status code. If found, it is served statically. If it is not found, Active4D returns the default (but very friendly) error message.

For example, to show a custom error page for a *404 (Not Found)* error, simply create an Active4D page and name it “404.a4d”, then place it in the root directory. When a 404 error occurs, Active4D will execute your custom page.

Note: If you use an executable HTTP error page to handle *401 Unauthorized*, which is the error status returned by the **authenticate** command, make sure the execution of the error page does not trigger another call of **authenticate**.

Within the context of an http error handler, the response status is 200 (OK). If your error handler does not change the response status, the original HTTP error status that triggered the error handler will be returned to the browser along with any output you generate.

CHAPTER 6

Invoking Active4D

If you are just starting to use Active4D, the Active4D shell provides everything you need to get up and running without the need to understand the details of the plugin API. If you are using the shell as is, you may safely skip this chapter.

If on the other hand you are integrating Active4D with your existing web server code, you will need to understand the plugin API.

Types of Execution

There are two different ways of executing scripts in Active4D: *request execution* and *direct execution*.

Request execution requires that you pass a valid HTTP request to Active4D. The request body contains the embedded script.

Direct execution requires only that you pass the embedded script to Active4D, although you may optionally pass a query string as well.

Request Execution

Once the host web server receives HTTP request, you pass the request to Active4D through one of the **A4D Execute <type> request** commands. They are:

A4D Execute BLOB request

A4D Execute 4D request

A4D Execute stream request

A4D Execute <type> request Parameters

The request execution commands all take similar parameters.

A4D Execute <type> request

version 2
modified version 3.0

A4D Execute <type> request(<inRequest>; inRequestInfo; outHeaderNames; outHeaderValues; outResponse) → Longint

<inRequest>	Varies	→	Request or means to retrieve it
inRequestInfo	Array Text	→	See below
outHeaderNames	Array Text	←	Response header field names
outHeaderValues	Array Text	←	Response header field values
outResponse	BLOB	←	Response body
Result	Longint	←	HTTP status code

Discussion

The array parameters must be text arrays, otherwise an error is generated.

The *inRequestInfo* array provides Active4D with context information not available in the HTTP request itself. It has a very specific format as follows:

Element	4DK# Name	Description
1	A4D Request Remote Addr	IP address of client (browser)
2	A4D Request Host Addr	IP address of server on which request was made
3	A4D Request Host Port	Port on which request was made
4	A4D Request Secure	"1" if SSL request, "0" if not

Examples of how to initialize this array are contained in the shell database methods *On Web Connection* and *A4D_HTTPD_RequestHandler* (NTK shell).

The <inRequest> portion of the parameter list changes depending on the command used. The parameters before the common parameters listed above are summarized in the following pages.

After handling a request, Active4D returns everything necessary to create a proper HTTP response in the last four parameters. The actual contents of those parameters depends

on whether or not the request executed successfully and what happened during execution.

outHeaderNames/outHeaderValues

If Active4D is given a request to an executable file, these parallel arrays receive the HTTP response headers. The headers returned for an executable file are as follows:

Name	Notes
X-VERSION	Always "HTTP/1.1"; always first element
X-STATUS	Always second element. See "Result" on page 68 for possible values.
Content-Type	The MIME type of the file
Content-Length	The size of the response body (outResponse)
Cache-Control	See RFC 2616 section 14.9
Pragma	Automatically added with the value "no-cache" if Cache-Control is "no-cache".
Expires	Unless explicitly set within Active4D, is set to the current time to prevent response caching
Location	If a redirect is performed
Set-Cookie	If a session is created and session cookies are enabled, or if a cookie is set within Active4D
WWW-Authenticate	If the authenticate command is used

In addition to these headers, you can set response headers within Active4D with the **set response header** command.

If Active4D is given a request to a non-executable file and is configured to serve non-executables, the following headers are returned:

Name	Notes
X-VERSION	Same as above
X-STATUS	Same as above
Content-Type	Same as above
Content-Length	The size of the file, which is in outResponse
Last-Modified	The file's last modified datetime in HTTP format
Cache-Control	If the 'nonexecutable expires' option in Active4D.ini is > 0, then this header is passed. For more info, see RFC 2616 section 14.9.

If Active4D is given a request to a non-executable file and is not configured to serve non-executables, *outResponse* is empty and the following headers are returned:

Name	Notes
Path	Full native path to the file
ModDate	Modification date of file in MM DD YYYY Forced format
ModTime	Modification time of file in HH MM SS format

Given this information, the host web server can easily serve the file. For example, using 4D's web server you could call **DOCUMENT TO BLOB** followed by **SEND HTML BLOB**. Using NTK you could call **TCP Send File**.

The modification date/time are returned to allow the host web server to implement a caching mechanism.

outResponse

This parameter will always contain the response body. The actual contents depends on the context:

Context	Contents
Executable file, successful execution	The result of the execution
Executable file, error occurred	Variable, see Chapter 7, "Interpreter," for more info
Non-executable file, no If-Modified-Since header passed in	The requested file (if it can be found)
Non-executable file, If-Modified-Since header passed in	If the file can be found and it has not been modified since the time passed in, the response body will be empty. Otherwise the response body will contain the requested file.

Result

The status code returned by Active4D will be one of the following:

Code	Name	Description
200	OK	All is well
301	Moved Permanently	A permanent redirect has been performed in response to a request from an HTTP 1.1 client
302	Found	A redirect has been performed in response to a request from an HTTP 1.0 client
303	See Other	A redirect has been performed in response to a request from an HTTP 1.1 client
304	Not Modified	A non-executable file is being served and the client passed an If-Modified-Since header and the file has not been modified since that time

Code	Name	Description
400	Bad Request	The request is malformed in some way
401	Unauthorized	You used the authenticate command
403	Forbidden	Either Active4D has timed out or an attempt was made to execute a script outside of the safe script directories
404	Not Found	The requested resource could not be found
408	Request Timeout	One of the callbacks returned an error during execution of <i>A4D Execute stream request</i>
413	Request Entity Too Large	The request size limit has been exceeded
500	Internal Server Error	An unrecoverable error has occurred during execution
-1	A4D Not Executable	A non-executable file was requested and the "server non-executables" option is off
-2	A4D License Timed Out	The continuous execution time period for the current license has expired

Active4D takes care of creating the proper response headers and response body for each status code.

A4D Execute BLOB request

version 2

A4D Execute BLOB request(*inRequest*; *inRequestInfo*; *outHeaderNames*; *outHeaderValues*; *outResponse*) → Longint

inRequest Text → Complete (usually) HTTP request
<standard params>
Result Longint ← HTTP status code

Discussion

This command expects the entire request (including uploaded files) to be in the *inRequest* parameter. It could be used with NTK if you retrieve the entire request into a BLOB first.

If 4D is running in Unicode mode, the contents of *inRequest* are assumed to be stored in the format *UTF8 Text without length*.

Note: If you are using NTK, I recommend using *A4D Execute stream request* instead of this command, as it performs a two-stage retrieval and relieves you of having to retrieve the request.

A4D Execute 4D request

version 2
modified v4.5

A4D Execute 4D request(inRequest; inRequestInfo; outHeaderNames; outHeaderValues; outResponse) → Longint

inRequest	Text	→	HTTP request
<standard params>			
Result	Longint	←	HTTP status code

Discussion

This command is designed for use with 4D built-in web server. It expects the header of the request to be in the *inRequest* parameter. For posted forms, the body of the request is retrieved through the *A4D_GetHttpBodyCallback* method.

If you want to change the callback method, call **A4D Set HTTP body callback**.

A4D Execute stream request

version 2

A4D Execute stream request(inStreamRef; inRequestInfo; outHeaderNames; outHeaderValues; outResponse {; outReqInfoNames; outReqInfoValues}) → Longint

inStreamRef	Longint	→	TCP/IP stream reference
<standard params>			
outReqInfoNames	Array Text	←	Receives request header names
outReqInfoValues	Array Text	←	Receives request header values
Result	Longint	←	HTTP status code

Discussion

This command is designed for use with an NTK-based web server. It does all the work of receiving the request from the TCIP/IP stream referenced by *inStreamRef*. It does so in two stages, first retrieving and parsing the request header, then retrieving the body if necessary.

A4D Execute stream request returns request info in the two parallel arrays *outReqInfoNames* and *outReqInfoValues*. The content of those arrays is as follows:

Name	Value
*url	The full requested URL
<HTTP header>	<HTTP header value>

The first element is the full requested URL with query parameters. The rest of the elements are exactly the same as the values returned by the Active4D **get request info** command (see “Request Info Collection Items”).

These arrays are provided to allow you to do custom post-processing of a request without having to parse it yourself. If you have no need for this information, you can simply not pass *outReqInfoNames* and *outReqInfoValues*.

Note: If Active4D is unable to read the request header, *outReqInfoNames* and *outReqInfoValues* will be empty.

Setting the Stream Callbacks

Before using this command you *must* call:

A4D Set stream callbacks(*inReceiveMethod*)

The parameter is a string which contains the name of the callback method used by Active4D to retrieve the request body from the TCP/IP stream.

Note: The callback method must be in the host database, not in a component.

The signature of *inReceiveMethod* is as follows:

ReceiveCallback

**version 2
modified v6.4r1**

ReceiveCallback(*inStreamRef*; *inStopString*; *inMaxLen*; *inTimeout*) → BLOB

<i>inStreamRef</i>	Longint	→	TCP/IP stream reference
<i>inStopString</i>	String	→	Receiving stops at this
<i>inMaxLen</i>	Longint	→	Maximum bytes to receive
<i>inTimeout</i>	Longint	→	Maximum ticks to wait
Result	BLOB	←	Received request body + status

Discussion

All of the parameters are passed in according to the rules of the NTK command **TCP Receive Blob**. Basically, you shouldn't have to worry about how this method works, since a receive callback method is provided for you in the shell.

The callback must append a single longint HTTP status code to the result BLOB.

Direct Execution

In addition to executing HTTP requests, there are also a set of commands that allow you to directly execute a file or to execute a block of text:

A4D Execute file

A4D Execute text

A4D Execute BLOB

With these commands you do not have access to most of the HTTP context information like cookies, form variables, etc., since there is no HTTP request to parse. You can, however, pass in a query string.

Uses for Direct Execution

Direct execution is useful in cases where you are not responding to an HTTP request. In fact, you could easily use Active4D as a general-purpose scripting engine by executing files or blocks of text that are nothing but Active4D code. You need only enclose the scripts in the `<% %>` tags.

A4D Execute file

version 2
modified version 3.0

A4D Execute file(*inPath*; *inQuery*; *inRequestInfo*; *outHeaderNames*; *outHeaderValues*; *outResponse*) → Longint

<i>inPath</i>	Text	→	URL-style path to file
<i>inQuery</i>	Text	→	Query string
<i>inRequestInfo</i>	Array Text	→	Execution context
<i>outHeaderNames</i>	Array Text	←	Response header field names
<i>outHeaderValues</i>	Array Text	←	Response header field values
<i>outResponse</i>	BLOB	←	Response body
Result	Longint	←	HTTP status code

Discussion

This command executes the file given by *inPath* as embedded Active4D code, just as if the file were being served as a web page.

If *inPath* begins with '/', it is considered an absolute path. If *inPath* does not begin with '/', it is considered relative to the default directory. For information on the default directory, see "The Default Directory" on page 42.

For more information on the format of *inRequestInfo*, see the discussion of *inRequestInfo* in "A4D Execute <type> request" on page 66.

A4D Execute text

version 2
modified version 3.0

A4D Execute text(*inText*; *inQuery*; *inRequestInfo*; *outHeaderNames*; *outHeaderValues*; *outResponse*) → Longint

<i>inText</i>	Text	→	Text to execute
<i>inQuery</i>	Text	→	Query string
<i>inRequestInfo</i>	Array Text	→	Execution context
<i>outHeaderNames</i>	Array Text	←	Response header field names
<i>outHeaderValues</i>	Array Text	←	Response header field values
<i>outResponse</i>	BLOB	←	Response body
Result	Longint	←	HTTP status code

Discussion

This command executes the text given by *inText* as embedded Active4D code, just as if the text were being served from a web page.

For more information on the format of *inRequestInfo*, see the discussion of *inRequestInfo* in “A4D Execute <type> request” on page 66.

A4D Execute BLOB

version 3.0

A4D Execute BLOB(*inText*; *inQuery*; *inRequestInfo*; *outHeaderNames*; *outHeaderValues*; *outResponse*) → Longint

<i>inText</i>	BLOB	→	Text to execute
<i>inQuery</i>	Text	→	Query string
<i>inRequestInfo</i>	Array Text	→	Execution context
<i>outHeaderNames</i>	Array Text	←	Response header field names
<i>outHeaderValues</i>	Array Text	←	Response header field values
<i>outResponse</i>	BLOB	←	Response body
Result	Longint	←	HTTP status code

Discussion

This command executes the text given by *inText* as embedded Active4D code, just as if the text were being served from a web page. It is assumed that the text is stored in the BLOB with the format *UTF8 Text without length*.

For more information on the format of *inRequestInfo*, see the discussion of *inRequestInfo* in “A4D Execute <type> request” on page 66.

CHAPTER 7

Interpreter

Much of the power of Active4D lies in its language. Unlike 4D's semi-dynamic HTML tags, which are a transmogrified *subset* of the 4D language, Active4D's language is a *superset* of 4D's language. In other words, with Active4D you just write 4D code. In fact, in some cases you can write and test code in 4D and then paste it into a text editor to use with Active4D.

Active4D's interpreter is actually quite a bit stricter than 4D's, in that syntax errors and nonsensical constructions that 4D blithely ignores (but the compiler catches, of course) are considered errors and abort execution.

Active4D extends 4D's language in some important ways. In addition to defining new operators, you can also define methods and libraries of methods. These are covered in Chapter 8, "Methods."

Flow of Execution

The Active4D interpreter is essentially a text processor. It parses text input (which might be a file or a block of text) and writes to a *response buffer*. The text input may have any mixture of HTML and embedded Active4D source code. When execution begins, the response buffer is empty.

Embedding Source Code

Active4D source is separated from HTML by one the processing tag pair `<% %>`.

Like 4D, Active4D is line-based; the end of a line or the end of a code block marks the end of a statement. However, you may have as much whitespace as you like between the Active4D tags and the actual source.

Input Parsing

Let's look at an extremely simple example which will illustrate how Active4D parses and executes a script. Suppose we execute a file which has the following contents:

```
<html>
I was here at <% write(current time) %>
on <% write(current date) %>
</html>
```

Active4D begins execution by scanning for a source code begin tag. As it scans, the text is appended to the response buffer. When the first “<%” in the above example is reached, the response buffer contains:

```
<html>
I was here at
```

Once a source code tag is found, Active4D skips past the tag and goes into execution mode. In execution mode, Active4D continues to parse the text, but instead of passing the text through to the response buffer, it resolves the text into executable tokens and then executes the tokens.

It would be pretty useless if you could not write dynamically generated text to the response buffer within your source code, so Active4D provides a **write** command to do just that. The **write** command takes a value of any type, converts it to text, and appends the result to the response buffer.

In the example above, the **write** command is the first token. The **write** command handler is given control, which proceeds to evaluate the parenthesized expression following the command. In this case the expression is the 4D command **Current time**, which is converted to text as if passed to the **String** command. The text is appended to the response buffer, and the interpreter then continues parsing.

The next token is “%>”, which is a source code end tag. Active4D skips to the end of the tag and switches back to scan mode. At this point the response buffer contains:

```
<html>
I was here at 19:27:13
```

The interpreter scans up to the next source code begin tag, appending to the response buffer. At the start of the next source code block, the response buffer contains:

```
<html>
I was here at 19:27:13 on
```

The interpreter then executes the second source code block as it did the first, after which the response buffer contains:

```
<html>
I was here at 19:27:13 on 10/19/2001
```

The interpreter scans to the end of the file and returns control to the caller. At that point the response buffer contains:

```
<html>
I was here at 19:27:13 on 10/19/2001
</html>
```

If the interpreter was invoked by Active4D’s web server, the web server generates the HTTP headers necessary to describe the response to the client. Finally, control returns to

4D, and the shell sends the generated headers and response body to the client browser, which displays the text:

I was here at 19:27:13 on 10/19/2009

The client cannot see the source code because it is stripped out before being sent back from the server.

Language Syntax

For the most part Active4D's language (henceforward just "Active4D") has the same syntax as 4D's language. However, there are a few important differences and enhancements.

English Only

When you write code in 4D's method editor, 4D looks up the command names and named constants in localized resources within the 4D application. For example, in English you might type "query" to do a database search, but in French the command name is "chercher".

After looking up a command or named constant, 4D stores a universal token to represent it. If you open the same method in different language versions of 4D, the tokens are displayed in the localized language.

Because Active4D code is written in raw text, it exists on disk in the form it was written. Hence no localization can be done, so all Active4D code must use the English 4D commands and named constants.

Source Code Structure

Because Active4D source code is plain text, you are free to use whatever indentation style and use of whitespace you wish — the 4D method editor is not there to do it for you. Typically you will want to use tabs in your text editor to perform indentation of control structures. For example:

```
<% write("hello") %>
<%
write("hello") `This is exactly equivalent to the previous block
%>

<%
  if ($sort_ascending = "1") // Note the space around the =
    order by([customers]; [customers]lastname; >)
  else
    order by([customers]; [customers]lastname; <)
  end if
%>
```

Since you *can* use whitespace in your code, it is highly recommended that you take advantage of the opportunity and get used to doing so.

You can have any number of Active4D code blocks within an HTML page. In addition, it is not necessary to complete control structures within the block in which they are declared. This powerful feature allows constructs like this:

```
<% for ($i; 1; 3) `I'm starting the loop here... %>
  Active4D rocks!<br />
<% end for %>

`Here is the output
Active4D rocks!
Active4D rocks!
Active4D rocks!
```

Using this feature with **If/Else/End if**, you can conditionally show entire tables, form items, etc.

Note: Just because your source is embedded inside your web pages in text form does *not* make it inherently insecure. To understand why, please see Chapter 4, “Security.”

Case Sensitivity

Like 4D, Active4D is not case sensitive, so you can just as easily write “first record” as “FIRST RECORD”. But whereas the 4D method editor would change “first record” to “FIRST RECORD” when it parsed the line, in Active4D it will remain “first record” because you will be using an external text editor.

Case-insensitivity extends to table and field names, method names, and variable names.

Expression-based

One difference between 4D’s language and Active4D is that Active4D is expression-based, like most computer languages. In other words, everything is an expression, including an assignment. So the assignment:

```
$i := 10
```

actually evaluates to the value of \$i after the assignment, namely 10. This allows you to do convenient tricks like this:

```
first record([contacts])

while (($name := [contacts]lastname) = "a@")
  writebr($name)
  next record([contacts])
end while
```

Comments

Active4D supports regular 4D comments (indicated by ```), either on a separate line or at the end of the line, with no limit on their length. You may also use the characters `"/"` as a synonym for ``` to indicate a 4D-style single-line comment. This style of commenting is common to C++, Java, and Javascript. Personally I prefer these comments to 4D-style comments, not only because I'm a longtime C++ programmer, but also because they are much more visible than tiny little backticks.

In addition to line comments, Active4D supports *block comments*. Unlike line comments, block comments may be embedded within a line or may span multiple lines.

Block comments begin with the character sequence `"/*` and end with the character sequence `*/`. For those of you who know other languages, this is the standard syntax for block comments in C/C++, Java, CSS, etc.

Active4D also supports *line continuation comments*. Line continuation comments are indicated by `"\"` and are the same as normal comments, except that they logically join the line on which they occur with the following line, allowing you to break a single line of code into multiple lines.

Here are some commenting examples:

```

/*
   This is a multi-line comment that will be rather lengthy.
   By using block comments I can format it easily and it is
   easier to read.
*/

`A plain old 4D-style single-line comment.
`It's kind of hard to see the stupid backtick.

// A C++-style single-line comment, much more visible
writebr("hello") // Also can be used at the end of a line

/*-----
   MyLibraryMethod($inFoo; $inBar) -> Text

   $inFoo Longint   A foo
   $inBar Longint   A bar
   RESULT Text      A foobar
-----*/

writebr("Block comments can even be " + /* wow! */ "embedded!")

/*
writebr("Using block comments...")
writebr("is a quick and easy way...")
writebr("to comment out a block of code")
*/

// Using line continuation comments
for ($i;      \\ Can have comment text
    $start;  \\ Useful for splitting up long parameter lists
    $end     \\
)
    writebr($i)
end for

```

Identifiers

Identifiers in Active4D follow the same *stated* rules as in 4D in terms of length and valid characters, with the following exceptions:

- The general rules for valid characters are the same, but the specific rules for valid Unicode characters can be found here:
[http://java.sun.com/j2se/1.3/docs/api/java/lang/Character.html#isJavaIdentifierStart\(char\)](http://java.sun.com/j2se/1.3/docs/api/java/lang/Character.html#isJavaIdentifierStart(char)).
- The maximum length for local variables, Active4D method names and library names is 255 characters.
- For backwards compatibility, table names may begin with a digit.

Active4D maintains its own local variables which are indicated by a leading dollar sign, as in 4D. You can also access existing 4D process and interprocess variables (using "<>", not the Macintosh diamond character), as well as any fields in the database. Last but not

least, all 4D 2004 and v11 named constants are available. If a 4D 2004 named constant was renamed in v11, both the old and new names are available.

Custom Named Constants

Active4D no longer has access to custom named constants you have defined. However, you can easily convert your custom constants to an Active4D library that will give you access to them. Please see the document “convert_constants.pdf” for more information.

Data Types

Active4D can operate on all 4D data types except for 2D arrays. Data types may be implicitly converted according to the same rules 4D uses, or explicitly converted using commands such as **String** and **Num**.

Compiler Declarations

All compiler declarations are supported, and may be used for local, process, and interprocess variables. For more information on declaring process and interprocess variables, see “Process/Interprocess Variables” below.

Active4D extends the 4D compiler declaration syntax by allowing you to use any valid scalar (non-array) expression, including dereferenced pointers and collection items. The collection item need not exist, in which case it will be created.

By predeclaring local variables with compiler declarations, you can achieve better type safety in your code.

For example:

```
c_longint($long)
$long := 7
$long := 13.27

/*
Without the compiler declaration, $long would become
a real = 13.27. Because it was declared a longint, its value
is now 13.
*/

$long := "foobar"    `generates an incompatible argument error
```

Active4D imitates 4D’s interpreted behavior exactly in regards to variables that have been declared in compiler declarations. Once a variable has had its type fixed by a compiler declaration there are two ways it can be changed:

- With another compiler declaration. If the new type is not assignment-compatible with the old type, the variable will be set to a null value. Otherwise a type conversion is done.
- By passing it as a parameter to a command that sets the parameter value inline, such as **GET PICTURE PROPERTIES**.

Process/Interprocess Variables

You may use compiler declarations to create process or interprocess variables on the fly, even in a compiled database. However, if your database is compiled, you may not change the type (or string width) of an existing process or interprocess variable.

Warning: Although you may create and reference process/interprocess variables within Active4D, you may *not* pass a pointer to those variables to 4D in a compiled database.

In addition to using compiler declarations, process and interprocess variables can be created on the fly through assignment, just as local variables are. For example, this works fine, even in a compiled database:

```
foobar := 7
get field properties(->[foo]bar; fieldType)
```

Of course, you should have a compelling reason to use process or interprocess variables instead of local variables. Basically, there are two main reasons for using process or interprocess variables in Active4D:

- You need to pass an array to a 4D method.
- You are using the **execute in 4d** command to execute 4D code, and the code needs to receive or return values.

If 4D is running in Unicode mode, there are no restrictions on process/interprocess variable usage. If 4D is running in compatibility mode, the following rules apply:

- Text variables/arrays are read write, whether they are declared in 4D or in Active4D.
- String variables/arrays appear to Active4D as text variables/arrays.
- If you declare a process/interprocess string variable/array in 4D, it is read only within Active4D.
- If you declare a process/interprocess string variable/array in Active4D, it is read write within both Active4D and 4D.

Array Support

All array types except 2D arrays are fully supported in Active4D. This includes picture and pointer arrays. You may declare and use arrays in Active4D just as you would in 4D. In most cases you should use local arrays, unless there is a compelling need to use process or interprocess arrays.

Note: Active4D's array support is *dramatically* expanded beyond that provided by 4D. You should take the time to carefully read the command section "Arrays" on page 141.

Pointer Support

Active4D supports pointers to process and interprocess variables, tables, and fields. You may create and dereference pointers to any of these entities within Active4D.

Unlike 4D, you may not create a pointer to a local variable. However, Active4D allows you to pass local variables *by reference* to its own methods, which is effectively the same as using a pointer (but easier). For information on pass by reference, see Chapter 8, "Methods."

Extended Boolean Expressions

Active4D supports an extended form of boolean expressions that allows for a much more concise coding style. In Active4D, any data type can be treated as a boolean expression according to the following rules:

Data type	Boolean value
String/Text	True if Length() # 0
Longint/Real	True if # 0
Date	True if # !00/00/00!
Time	True if # ?00:00:00?
BLOB	True if BLOB size() # 0
Picture	True if Picture size() # 0
Pointer	True if Not(Nil())
Array	True if Size of array() # 0

This allows you to write code like this:

```
if ($text)
  write($text)
else
  write("Text is empty")
end if

array longint($longs; *; 7; 27)
write(choose($longs; "Have values"; "Empty array"))
```

Note: Extended boolean expressions can only be used with commands and keywords that take a boolean expression, not the logical operators & and |.

Literals

Active4D recognizes string, number date and time literals in all the standard formats, including scientific notation. Note that numeric literals may contain your locale-specific

grouping separators. For example, the number 1 million can be typed as "1,000,000" in the US and "1.000.000" in Germany.

Note: Time literals must use '?' as the delimiter.

String Literals

Like 4D, all strings literals support backslash-prefixed embedded control characters:

Backslash combination	Resolves to
\n	Char(Line feed)
\r	Char(Carriage return)
\t	Char(Tab)
\"	Char(Double quote)
\'	Char(Quote)
\\	\

If you have not yet gotten used to using backslashed characters, you should definitely get used to using them instead of the equivalent **Char(X)** expression.

Here is an example of using backslashes in a literal, with the equivalent **Char(X)** code. You decide which is easier to write and read.

```
write("This is line 1.\r\"It's cool!\", you say.")

Output:
This is line 1.
"It's cool!", you say.

// Pre-4D 2003 code to do the same
"This is line 1."+Char(13)+Char(34)+"It's cool!"+Char(34) \\
+", you say."
```

However, there is a better way to embed double-quotes in a string literal. Active4D also supports string literals surrounded by single quotes, within which double-quotes are just another character. So the example above becomes:

```
write('This is line 1.\r"It's cool!", you say.')
```

String Interpolation

How many times have you had to write a line of code like this:

```
writeln("Name: " + [contacts]name + " (" + $formattedPhone + ")")
```

We are used to writing code like this, so it may seem normal. But the problem is that all of the + operators obscure the actual format of the string you are trying to write. Wouldn't it be nice if you could do something like this:

```
writeln('Name: [contacts]name ($formattedPhone)')
```

Now it is very clear what the final string will look like. The good news is that you can do exactly what is shown above by using *single quotes* instead of double quotes to delimit the string.

There is an additional property of single-quote delimited strings that adds a very powerful feature. This feature is called *string interpolation*. Basically, what string interpolation means is that the string is parsed for embedded values and those values are replaced within the string. The types of embedded values recognized are:

- **Variable references:** Local variable names, followed by one or more array or collection indexes, may be directly embedded within a string.
- **Built-in collections:** References to values in the *_form*, *_query*, *_request*, *globals* and *session* collections.
- **Field references:** Full *[table]field* references. If the field name contains spaces, it must be treated as an expression.
- **Arbitrary expressions:** Arbitrary executable expressions.

Note: Interpolated strings may *not* contain nested indexes, such as:

```
session{$keys{$i}}
```

To understand the power of this feature, we need to see some examples. First, here is a simple embedding of a variable within an interpolated string:

```
$name := "Mr. Phelps"
$msg := 'Good evening $name. Your mission...'
// $msg contains "Good evening Mr. Phelps. Your mission..."

write('$name is a swell guy.')
// output: "Mr. Phelps is a swell guy."
```

If we want to reference a form variable or query parameter, use this style:

```
write('Thank you for registering, _form{"f_name"}.')
write('The id to edit is _query{"id"}.')')
```

To reference a field, use this style:

```
write('Thank you, [Contacts]Firstname!')
```

Okay, that was simple enough. What if we have an array? No problem:

```
<%
array string(80; $names; 0)
array longint($ages; 0)
// set arrays somehow
%>

<table>
  <tr>
    <td>Name</td><td>Age</td>
  </tr>
<%
for ($i; 1; size of array($names))
  writeln(' <tr>\n<td>$names{$i}</td><td>$ages{$i}</td>\n</tr>')
end for
%>
</table>
```

Note in the example above that we referenced a longint. All types that can be converted to text are automatically converted. If you try to embed a variable expression that is not convertible to text, it will generate an error.

What if we want to embed a collection reference? Again, no problem:

```
$info := new collection("name"; "Dave"; "age"; 27)
array string(31; $info{"children"}; 0)
set array($info{"children"}; "Jody"; "Buffy")

writebr('Name: $info{"name"}, age: $info{"age"}')

for ($i; 1; size of array($info{"children"}))
  writebr('Child: $info{"children"}{$i}')
end for

write('First letter of name is "$info{"name"}[[1]]"')
```

Basically, you can embed any valid variable reference, with an optional character reference at the end.

What happens if the variable or field name or *_form/_query/_request* reference is followed or preceded by valid identifier characters? How do we distinguish between the embedded reference and the surrounding text? In a case like this you have to enclose the expression in backticks (`), like this:

```
$verb := "sleep"
write('Are you $verbing?') // error: unknown variable $verbing
write('Are you ` $verb `ing?') // outputs "Are you sleeping?"
```

In actual fact, anything enclosed in `` is evaluated as an Active4D expression, as if you were assigning the result of the expression to a variable. This allows you to embed arbitrary expressions, like this:

```
write('It is `current time` on `string(current date; long)`')
```

You can use anything in the expression, including method calls, but the expression must return a result or an error will be generated.

If you need to use one of the reserved characters within an interpolated string, simply prefix it with a backslash to prevent it from being interpolated, like this:

```
$amount := 100
write('You owe me \$$amount.')
// output is "You owe me $100."
```

Heredoc Strings

Active4D also allows you to define strings literals that span multiple lines. Such string literals are known as *heredoc strings*. A regular heredoc string begins and end with the character sequence `"""` (three double quotes). An interpolated heredoc string begins and ends with the character sequence `'''` (three single quotes), and is interpolated as described above for interpolated strings.

As with interpolated strings, the big advantage of heredoc strings is clarity: in your code you can see multi-line strings as they will appear in the final output.

Here is an example:

```
$cell := '''
[Contacts]Firstname [Contacts]Lastname<br />
[Contacts]Address<br />
[Contacts]City, [Contacts]State [Contacts]Zip<br />'''
writeln($cell)

// output will be something like this:
Homer Simpson<br />
123 Main St.<br />
Springfield, MO 12345<br />
```

Heredoc allows you to do complex multiline formatting in a very natural style, with all of the power of interpolated strings if you so desire.

Date Literals

Date literal parsing is more rigid than in 4D. Dates must contain three parts consisting of digits, separated by a forward slash, space, dot or dash.

Time Literals

Time literals may use spaces as separators in addition to colons, and each part of the time (hours/minutes/seconds) may contain a single digit.

User-defined Constants

In addition to supporting 4D named constants, Active4D allows you to define named constants at runtime using the **define** command. For more information on the **define** command, see “define” on page 242.

Typing of Values

Typing of local variables in Active4D follows the same rules as 4D. If a local variable has not been declared with a compiler directive, it is variant: its type changes on the fly according to the type of value assigned to it. Once it has been declared within a compiler declaration, it follows the rules outlined in “Compiler Declarations” above.

On the other hand, 4D variables and fields are always considered invariant. You cannot assign a value to them that would change their type.

For all variables and fields, Active4D will generate an error and abort execution if a referenced variable or field is undefined.

Operators

Active4D implements all of 4D’s operators (with the exception of some picture operators), including array indexing and character references.

Active4D operators have the same restrictions on the data types they will accept as 4D does. Active4D will abort with an error if an attempt is made to divide or modulo by zero.

Unary/Assignment Operators

Active4D adds many unary and assignment operators (borrowed from other languages) which make writing code faster and easier to read. Here are the new operators:

++	prefix and postfix increment
--	prefix and postfix decrement
::=	super assign
+=	add and assign
-=	subtract and assign
*=	multiply and assign
%=	modulo divide and assign
/=	divide and assign
\=	integer divide and assign
^=	exponentiate and assign

|= bitwise inclusive OR and assign
 &= bitwise AND and assign
 <<= bitwise shift left and assign
 >>= bitwise shift right and assign

Here are some examples of their usage:

```

$i := 3      // I think $i=3 now
$j := ++$i  // increments $i before the assignment, $j=4
$k := $i++  // increments $i after the assignment, $k=4, $i=5
$i *= 3     // multiplies $i by 3 and assigns, $i=15
$i += 3     // adds 3 to $i and assigns, $i=18
$i >>= 1    // bit shifts $i one bit and assigns, $i=9
  
```

Any expression:

```
<assignable> <op>= <value>
```

is equivalent to:

```
<assignable> := <assignable> <op> <value>
```

Any of these operators will work with array elements. So, for example, you can increment the fifth element of an array by using the ++ operator:

```

$myArray{5}++
++$myArray{5}
  
```

Super Assign

Active4D adds a new operator called “super assign”, represented by “::=”. Super assign is like the normal := operator but can only be used to assign to variables or collection items, and completely replaces the original variable’s value with a copy of the value to the right of the operator. This allows you to copy arrays by assigning to a variable directly.

For example,

```

array text($array; *; "one"; "two")
$copy ::= $array
  
```

is equivalent to:

```

array text($array; *; "one"; "two")
copy array($array; $copy)
  
```

The advantage of using super assign is that it allows you to write generic code that copies values without special-casing for arrays.

In and Not In Operators

Active4D adds two comparison operators that make looking for a value in a sequence much easier and consistent across all sequence types, where a sequence can be a collection, an array, or a string.

```
~      in
!~     not in, opposite of ~
```

In general, every time you say to yourself, “Is this value (not) in that sequence,” you can use the `~` and `!~` operators. The examples below demonstrate how to do *in* matching using the `~` operator and the equivalent code using PO4D (Plain Old 4D):

```
if ($name ~ $contactNames) // using ~
if (find in array($contactNames; $name) > 0) // PO4D

if ($sub ~ $text) // using ~
if (position($sub; $text) > 0) // PO4D

if ("b_cancel" ~ $attributes) // using ~
if (collection has($attributes; "b_cancel")) // PO4D
```

Regular Expression Operators

Active4D also adds two comparison operators (borrowed from Perl) that make regular expression matching an integral part of the language:

```
=~     regular expression match
#~     regular expression non-match, opposite of =~
```

Using these operators is exactly equivalent to using the **regex match** command:

```
// this...
if ($name =~ "/(Dave|John|Bill)/")

// ...is the same as this
if (regex match("/(Dave|John|Bill)/"; $name))

// and this...
if (not($name =~ "/(Dave|John|Bill)/"))

// ...is the same as this
if ($name #~ "/(Dave|John|Bill)/")
```

For more information on regular expression matching, see “[regex match](#)” on page 281.

String Format Operators

As was discussed above, Active4D adds string interpolation as a simple way to embed placeholder values and expressions within a string which are replaced at runtime.

Active4D offers two other powerful alternatives for interpolating strings, using two special string formatting operators: %% and %.

The %% operator takes a *format string* on the left. The format string specifies the number and type of the *format arguments* that are to the right of the %% operator. If there is more than one format argument, they must be enclosed in parentheses and separated by semicolons. At runtime the format arguments are converted and inserted into the format string according to the syntax of the format string.

For more information, see “%% (formatting operator)” on page 364.

The % operator has three meanings in Active4D:

- If it is used as a comparator within a database query, it means to perform a keyword search, just as it does in 4D.
- If it is preceded by a number, it means modulo division, just as it does in 4D.
- If it is preceded by a text expression, it follows the same syntax as the %% operator.

For more information, see “% (formatting operator)” on page 363.

Picture Operators

Active4D supports horizontal and vertical concatenation for pictures. Thus the operators that may be use with pictures are:

+	horizontally concatenate
/	vertically concatenate
+=	horizontally concatenate and assign
/=	vertically concatenate and assign

Pointer Dereference Operator

The pointer dereference operator (->) works as usual when applied to a pointer. When applied to a collection reference, it allows you to treat the collection as a kind of object and call methods for that object. For more information, see “Creating a Poor Man’s Class” on page 124.

+ Operator

The + operator has been enhanced in Active4D to auto-convert arguments to text if the argument one of the arguments is text.

For example:

```
$s := "I was here at " + current time + " on " + current date
```

Notice there is no need to explicitly convert **Current date** and **Current time** to strings.

The argument to the left of the + will also be converted to text if the argument to the right is. For example, this will work:

```
$s := current date + ", " + current time + ": I was here"
```

The following value types will be auto-converted:

Type	Format
Longint	4D default
Real	4D default
Date	4D default
Time	4D default
Boolean	"True" or "False"
Pointer	->[table] ->[table]field ->variable

Character Reference Operator [[]]

The character reference operator has been enhanced in Active4D to allow you to specify a negative number. In that case the number represents the Nth character from the *end* of the text. So `$text[[-1]]` will return the last character of `$text`, `$text[[-2]]` will return the second to last character, and so on.

For example:

```
$text := "Active4D"
$c := $text[[-1]] // $c = "D"
$c := $text[[-2]] // $c = "4"
```

Indexing Operator {}

Active4D extends the syntax of array indexing in two convenient ways.

First, there is a super easy way to add elements to an array. If you use an empty index with an array, a new element is appended to the array and the index is set to the newly appended element.

This allows you to add items to an array in a very simple way, like this:

```
array longint($longs; 0)
$longs{} := 7 // same as append to array($longs; 7)
$longs{} := 13 // same as append to array($longs; 13)

// $longs now contains 2 elements, 7 and 13
```

This syntax works anywhere you can use an array element, not just as the target of an assignment. For example, you can do things like this:

```
array longint($types; 0)
$stable := table(->[vendors])

for ($field; 1; count fields($stable))
  get field properties($stable; $field; $types{})
end for

// $types is now filled with the [vendors] field types
```

The second extension to the array indexing syntax is the addition of negative indexes. By using a negative index, you index array elements from the end of the array, with -1 being the last element and *-Size of array* being the first element.

For example, to reference the last element of an array, you can simply do this:

```
$myArray{-1} := 7

// old way, which one is easier?
$myArray{size of array($myArray)} := 7
```

Boolean Operator |

The boolean | operator may be used with text. If the operand on the left is empty, the expression resolves to the operand on the right, else to the operand on the left. This is useful for avoiding the common idiom:

```
if ($attributes{"nm"} = "")
  $foo := "Some default value"
else
  $foo := $attributes{"nm"}
end if

// using choose
$foo := choose($attributes{"nm"} # "" ; $attributes{"nm"} ; \\  
"Some default value")
```

The above code can now be replaced with the much clearer and more concise:

```
$foo := $attributes{"nm"} | "Some default text"
```

Control Structures

All of 4D's control structures and flow of control keywords are supported, with four notable additions: **for each/end for each**, **break**, **return**, **exit** and **continue**. If you are familiar with other languages that use these keywords, they work exactly as they do in those languages. If you aren't familiar with those languages, here's how they work.

for each/end for each

This looping control structure is the easiest way to iterate over a sequence of values, which includes collections, strings, arrays, and selections. For more information, see “for each/end for each” on page 245.

break

If you are within any kind of loop (**For**, **for each**, **While**, **Repeat**), the **break** keyword (used on a line by itself) will transfer execution to the first line of code after the end of the loop. The loop variable used by a **For** loop will not be changed. If you use **break** outside of a loop it will generate an error.

return

This keyword (used on a line by itself) transfers execution to the first line of code after the current code block. If you are within an Active4D method, execution will continue at the first line of code after the line that called the method. If you are within an included file, execution continues after the include statement that included that file.

return can also take a parenthesized expression to return as the result of the method.

continue

If you are within any kind of loop (**For**, **for each**, **While**, **Repeat**), the **continue** keyword (used on a line by itself) will transfer execution directly to the top of the loop.

- The loop variable in a **For** loop will be incremented according to the **For** clause.
- The next item will be fetched in a **for each** loop.

If you use **continue** outside of a loop it will generate an error.

exit

This keyword, used on a line by itself, immediately aborts all execution without generating an error. This is primarily useful for debugging, when you want to dump the internal state before a certain point and then stop. This keyword is also useful if you have detected an error condition from which you cannot recover and you want to immediately stop execution.

Examples

Here are some examples of how to use the new keywords:

```
// Using break to terminate an infinite loop
$i := 0

while(true) // You could never do this in 4D!
  if (++$i > 10)
    // The closest loop, in this case the while loop,
    // will be exited
    break
  end if

  write($i) // This will NOT be executed once break is executed
end while

// Here's the Active4D way of breaking out of a for loop

for ($i; 1; size of array($names))
  if ($names{$i} = "g@")
    break
  end if

  // This will not get executed after break is executed
  writebr($names{$i})
end for

// Here's the 4D way

for ($i; 1; size of array($names))
  if ($names{$i} = "g@")
    $i := size of array($names)+1
  else
    writebr($names{$i})
  end if
end for

// Using continue in a for loop

for ($i; 1; size of array($names))
  if ($names{$i} = "s@")
    continue // Immediately goes to next iteration of loop
  end if

  writebr($names{$i})
  // Do lots of other stuff here.
  // The effect of continue above is to skip this
end for
```

Working with Paths

In the course of programming a web site with Active4D, you will often need to specify file path. If you have done any work with paths in 4D, you know what a pain it can be, especially when dealing with multiple platforms.

URL-Style (Posix) Paths

All commands that take a path in Active4D — including Active4D’s implementation of standard commands like `cd` — can take a URL-style (Posix) path, which uses `’/’` as the directory separator. This allows you to program in a platform-neutral way, without having to worry about the native directory separator.

Absolute vs. Relative Paths

A path can be *absolute* or *relative*. What that means depends on the command using the path. Standard 4D document commands and Active4D’s own commands treat paths differently.

- **4D document commands:** As with 4D, in Active4D absolute paths are relative to the computer, allowing you to access any volume mounted on the host machine (including network volumes). Remember, however, that by default Active4D restricts document command access to the web root directory. You must use the “safe doc dirs” option in Active4D.ini to gain access to directories outside the web root directory. For more information, see “The “safe doc dirs” Config Option” on page 51.

As with 4D, relative paths are relative to the default directory (see “The Default Directory” on page 42).

- **Active4D commands:** Active4D commands are designed for use within the context of a web page. Thus their “world,” so to speak, is limited to the web root directory.

Absolute paths used with Active4D commands are relative to the web root directory. Thus if the root directory is:

```
/Users/tom/db/web
```

the path

```
/accounting/default.a4d
```

is actually the path

```
/Users/tom/db/web/accounting/default.a4d
```

Relative paths used with Active4D commands are relative to the directory of the currently executing file.

Path Utilities

Active4D provides a many utility commands for working with paths. For example, if you need to use a 4D document command on a file within the web root directory, the **get root** command returns the full path to the current root. In addition, there are commands to extract the filename from a path, the extension from a filename, and the directory

from a path. If you need to join path segments together, you can (and should) use the **join paths** command.

For more on these commands, see “System Documents” on page 387.

Path Limits

The host operating system imposes a maximum length on paths and components of a path.

On macOS the maximum length of a single component of a path (file or folder name) is 255 bytes. The maximum length of a path is 1024 bytes. Note that these limits are in bytes, not Unicode characters, and that paths are encoded in UTF-8. Non-ASCII characters in a path will take 2-4 bytes each, reducing the maximum number of Unicode characters accordingly.

On Windows the maximum length of a single component of a path (file or folder name) is 255 Unicode characters. The maximum length of a path is 260 Unicode characters.

You must ensure that your path lengths are within these limits, or Active4D will not be able to find your files.

Including Other Files

One of the most powerful features of Active4D is its ability to include other files during execution. To include a file, you use the **include** command, passing a relative or absolute URL-style path. Since **include** is an Active4D command, absolute paths are relative to the root, and relative paths are relative to the currently executing file.

If found, the included file is interpreted as if it were part of the source file. The scope of the included file is whatever the scope was where the **include** command appeared.

This means that any local variables declared before the include are available in the scope of the include file’s code, and any local variables declared in the include file are available to the source file when the include file is finished executing.

Included files may in turn include other files, ad infinitum (or until stack space or memory runs out).

Note: You may also use the **include into** command, which allows you to include a file and place the output into a variable.

Uses of Included Files

There are many uses of included files. Some common ones are:

- Factoring out common page elements, such as headers and footers. When the included file is modified, all pages that include it automatically update.
- Separating functional sections of a page into separate files. This allows you to create a page design that consists of an overall framework, within which you “plug in” various pieces. By using includes, you can edit the pieces separately without affecting the

overall design. This technique is analogous to splitting a large method into several smaller methods.

- Conditionally building a page. You may put an **include** statement within an **If/Else/End if** or **Case/End case** construct to conditionally include various files. This allows you to use the same page to display different output based on one or more criteria.
- Using **include into** to build an HTML page to send via email.

Including Only Once

You can also include files using the **require** command. The **require** command works like **include**, but it guarantees that any given file will only be executed once via **require** within a single execution of the interpreter.

The **require** command is primarily useful for creating files of global constants, variables and methods that only need to be loaded once per Active4D session. All files that reference these globals can **require** the globals file and you don't have to worry about the overhead of executing the file or the problem of defining named constants twice, which is an error.

Calling 4D Methods

Although you can call 4D methods within Active4D, this ties you to the database structure, which is exactly what Active4D is designed to avoid. If possible, you should define and use methods within Active4D. For more information on defining methods in Active4D, see Chapter 8, "Methods."

Nonetheless, should you need to, you can call any 4D method within Active4D using the exact same syntax you would within 4D.

Parameter Passing

When calling a 4D method from Active4D, you pass parameters just as you would within 4D. Strings are converted to Text and Longints are converted to Reals before being passed. You may return any type of value from the method.

Warning: If you are passing textual parameters to a 4D method from Active4D, the parameter in 4D must be declared C_TEXT. If you are passing numeric parameters, they must be declared C_REAL. Otherwise a runtime error may occur in a compiled database.

Active4D has no way of matching parameters passed to a 4D method with the actual parameters declared in that method. It is up to you to make absolutely sure that the number and type of parameters declared in the 4D method are compatible with what you pass to it. If the 4D method expects some parameters to be optional, you may of course not pass those.

In an interpreted database, if the method parameters are not assignment-compatible, 4D will initialize the parameter to a null value. In a compiled database, however, if the

parameter types do not match exactly, a runtime error will be generated, which will effectively bring your application to a halt in a very unfriendly way. You do not want this to happen.

Indirect Method Calls (aka Poor Man's method pointers)

In addition to calling 4D methods as you would within 4D, by directly referencing the method name, you can also indirectly call a method by name using the **call 4d method** command.

The syntax of this command is as follows:

```
call 4d method(inMethodName {;inParam1 {;inParamN}})
```

The method name may be any valid expression that returns text. This powerful feature allows you to dynamically determine which of several methods you call, as long as their parameter lists are compatible. This is (sort of) the equivalent of a method pointer in other programming languages.

As with ordinary method calls, the method called by **call 4d method** may return a value.

If the 4D method returns a value, you may ignore it if you have no use for it. You need not assign it to a dummy variable as you would in 4D. On the other hand, if the 4D method returns no value, attempting to use the result of such a method will result in an error.

Collections

Active4D adds a new data type to the language: *collections*. A collection is a group of *key/value pairs* which are stored in memory. The keys are strings (with a maximum length of 2GB) and the values may be any 4D data type, including arrays. The key/value pair is also referred to as an *item*.

In classic programming terms, a collection is also known as an associative array, a dictionary, a symbol table or a map. In 4D terms, you can conceptually think of a collection as two parallel arrays, with keys being in one array and the values in the other. Of course, you can't do this in 4D because 4D arrays can only contain a single type, and a collection value may be of any type.

Active4D provides a full suite of commands for creating your own collections. In addition, Active4D uses collections to store HTTP headers, query parameters, form variables, cookies, global variables, and sessions.

For more information on the collection commands, see "Collections" on page 153.

Note: In previous versions of Active4D, collection keys were kept in alphabetical order, although officially no order was guaranteed. In v6 the keys are kept in no predictable order, so if you have code that relies on alphabetical keys, you will have to get the collection keys, sort them, and then use the sorted keys to access the collection items.

Collection Handles

Collections are referred to by a *Longint* handle, much like an ObjectTools object handle or a 4D hierarchical list handle.

Active4D maintains an internal list of user-created collections. All collection commands that take a collection handle check the handle against this list and generate an error if the handle is not valid. Thus you are prevented from crashing the server by passing in a bogus handle.

Local vs. Global Collections

Collections can be either *local* or *global*. A local collection is automatically deleted when a script finishes executing. A global collection remains in memory throughout the life of the server, or until it is specifically deleted.

Local collections are like local variables — you use them for temporary storage within a single script. You should *always* use a local variable to store a local collection handle.

Any time you want a collection to outlive the current script execution, you must use a global collection.

Note: Once you create a global collection, it is your responsibility to delete it when it is no longer needed.

Using Collections

Collections come in two basic varieties: *read-only* and *read-write*. You can perform the following operations with read-only collections:

- Get a collection value given a key. If the value is an array, you may retrieve an element of the array directly. Key matching is case-insensitive.
- Get all keys into a 4D array.
- Get all values into a 4D array if it is known they are all of the same type.
- Get the count of key/value pairs in the collection.

Read-write collections add the following operations:

- Set a collection value given a key. If the value is an array, you may set an element of the array directly.
- Delete a collection item given a key. If the key contains a wildcard, all matching items are deleted.

In addition to these basic operations, some of the specialized collections defined by Active4D provide other operations as well. These operations are covered in the relevant command reference sections.

Referencing Collection Values

Active4D extends the syntax of the `{}` indexing operators to allow indexing a collection by keys. The syntax of this way of indexing is as follows:

```
collectionRef{key}
```

where *collection* is either a collection handle or collection iterator, and *key* is a text expression. If an item in the collection exists with the given key, the result of this expression is the item's value, and it may be treated in all respects as a variable of that type. If no item exists in the collection with the given key, the result of the expression is an empty string.

Because the result of the expression is treated as the value it resolves to, you can use collections as a natural part of the language. For example:

```
// Set a counter in our session
session{"counter"} := 0

// Now increment the counter
++session{"counter"}

// Store a form variable in the session
session{"username"} := _form{"f_username"}

// Embed an array in a collection
$c := new collection
array text($c{"people"}; *, "Tom", "Dick", "Harry")

for ($i; 1; size of array($c{"people"}))
  writebr('Hi $c{"people"}{$i}')
end for

$c{"people"}{} := "Louise"
```

Embedded Collections

You can embed collections in collections (by storing their handles) to any depth and reference their items by adding more indexes, like this:

```
$foo := $c{"level1"}{"level2"}{"level3"}{"the_key"}
```

You may safely embed global collections within a local collection, because the global collections will outlive the local collection in which they are embedded. On the other hand, you must embed *only* global collections within other global collections to ensure the embedded collection is valid throughout the lifetime of the containing collection.

Element Referencing

If when referencing a collection array element, you use the form:

```
collectionRef{key}{index}
```

If the item is in fact an array, you may use any value from 0..size of array for the *index*. If the item is scalar (not an array), you may still use the index form above, but the index must evaluate to 1.

This form allows you to reference a collection item with the same syntax whether or not it is an array. This feature is mainly designed for use with multiple-choice form lists, which may result in a scalar value (if only item is selected) or an array value (if multiple items are selected).

Iterating Over a Collection

Very often you may need to iterate over every item in a collection. There are two ways to do so in Active4D.

The first (and easiest) way to iterate over a collection is to use the **for each/end for each** loop control structure. For more information on **for each**, see “Iterators” on page 214.

The second way to iterate over a collection is through an *iterator*. Every collection provides a command which returns an iterator for the collection. You use this iterator to traverse the items in the collection.

In addition, some collections allow you to get an iterator for a specific item given the item’s key. If no item with the given key exists, you are given an *empty iterator*. An empty iterator is a special iterator that has the following properties:

- The iterator itself (a Longint) is zero
- **is an iterator** will return *False*
- **get item key** will return an empty string
- **get item value** will return an empty string
- **get item type** will return *Is Undefined (5)*

You can identify an empty iterator either by testing equal to zero, by calling **is an iterator**, by testing for an empty key, or by testing the item type.

For more information on iterators, see “Iterators” on page 214.

HTTP Data Access

When Active4D is used as the web server, the interpreter has full access to both the HTTP request and the response data. This data is critical when developing high-powered web sites. The commands necessary to access this information are covered in Chapter 11, “Command Reference.”

Request Data

As was discussed in Chapter 5, “HTTP Server,” an HTTP request consists of several headers and an optional body, in addition to the requested URL itself.

The data encapsulated in an HTTP request includes:

- Query string parameters
- Form variables
- Cookies
- HTTP headers
- Uploaded files

In addition, information about the host environment is passed in to Active4D.

It is important to note that all of this data is encoded in some way or another. Without Active4D, to extract any meaningful information you would have to:

- 1 Understand the detailed HTTP specification and format for each type of data.
- 2 Write the code to parse and extract the data, making sure to follow the rules you learned in Step 1.
- 3 URL-decode the data which, according to the HTTP specification, should be URL-encoded.
- 4 Convert the URL-decoded data to Unicode.
- 5 Figure out where and how to store the data in a meaningful way.
- 6 Write many methods to access that data in a simple way.

If you have never gone through this process, here’s a little tip: the time it takes to do Step 1 alone will cost you more than the price of Active4D!

Fortunately Active4D does all of the above for you. You never have to deal with the particulars of the HTTP specification, which allows you to focus on the problem at hand — building a great web site.

_query and _form Collections

The primary way in which you “pass” parameters from one page to another in a web site is through form variables and query string parameters. Active4D places those parameters in easy-to-access collections.

For example, if the user posts a form which contains a field called “f_name” and you want to access the contents of that field, you can simply use:

```
_form{ "f_name" }
```

Likewise, to access a query string parameter called “recnum”, you could use:

```
_query{ "recnum" }
```

Testing Form Buttons

If a non-image submit button is clicked on a form, the browser posts the button's form name and value. Buttons which appear on a form but are not clicked are not included in the posted form data.

Frequently you need to test a posted form to see which button was clicked. For example, your form may have "Search", "Previous" and "Next" buttons. Because non-existent collection items are returned as empty strings, you can quickly test for which button was clicked by checking for empty strings. For example, your code would look something like this:

```
case of
  :(_form{"f_search"} # "")
    // Do the search
  :(_form{"f_previous"} # "")
    // Go to the previous group of records
  :(_form{"f_next"} # "")
    // Go to the next group of records
end case
```

Response Data

You control the response body indirectly with the **write** command and its peers. In addition, Active4D gives you dedicated commands for setting the response headers.

The data encapsulated in an HTTP response includes:

- Cache control
- Character set
- Content type
- Cookies
- Expires date
- Other headers

Some HTTP response headers are simple in their format. Others require a specific format which you must follow. As with the request headers, without Active4D you would have to know the HTTP specification for each format. For those headers that require special formatting, Active4D gives you simple commands that relieve you of having to know the HTTP specification.

Working with Character Sets

Internally Active4D (like 4D) uses Unicode exclusively. Text comes into Active4D from several sources, each of which may be in a different character set (or "charset"):

- **Executable source files:** These may be in any charset, but if you are working with a non-Asian language, it is recommended you use UTF-8.
- **Database:** 4D uses Unicode internally and expects text to be in this charset.

- **URL query strings:** Query strings are URL-encoded UTF-8.
- **Posted form data:** Form data is URL-encoded in the charset of the page in which the form appears.
- **Files read programmatically:** Text files may be in any charset. It is up to you to know which one.

In each case, text must be converted to Unicode.

Text can go out from Active4D to several destinations:

- **Web browser:** Text sent to the browser via the **write** commands should be in the target HTML charset, usually UTF-8. In addition, you may need to HTML-encode reserved characters such as '<'.
- **Database:** 4D expects Unicode.
- **Files written programmatically:** You can use whatever charset you want.

Clearly, it would be a real pain — not to mention error-prone — if you had to remember to do all of the character set conversions yourself. Fortunately, Active4D allows you to configure the various charsets and then takes care of most of these conversions for you.

Internally Active4D uses Unicode, since that is what 4D expects and because it allows Active4D to work with any character set on Earth. The character sets you configure determine which charset Active4D converts *from* on input and which charset it converts *to* on output.

Platform Character Set

The platform charset determines what charset Active4D converts *from* when reading executable source files, which may contain non-ASCII characters both in HTML and in Active4D string literals.

For example, if you are using a programmer's editor to write your embedded scripts, identifiers and string literals would most likely be in UTF-8. On the other hand, if you are writing your scripts with Dreamweaver, string literals would be in the charset of the page you are creating.

You can set the platform charset with the "platform charset" config option in Active4D.ini. You may also use the **set platform charset** command. For more information on the "platform charset" option, see the comments in Active4D.ini.

Warning: If your source files contain non-ASCII characters, it is essential that they *all* are encoded in the same character set, and that character set is configured as the "platform charset" in Active4D.ini. Otherwise those non-ASCII characters will be incorrectly converted to Unicode when the source file is read by Active4D.

On a Japanese language system, the default platform charset is “shift_jis”. On a Chinese language system, the default platform charset is “gb2312”. For all other systems, the default is “utf-8”.

Note: In previous versions of Active4D, the default platform charset for non-Japanese/Chinese systems was “mac” on macOS and “windows-1252” on Windows. If your source files are not in UTF-8 (which they should be!) and you did not set a platform charset in Active4D.ini, you will have to set the platform charset now.

Output Character Set

Before returning to 4D, Active4D must convert its response buffer to a BLOB. If the response buffer is text, it must be converted from Unicode to the charset that will be sent to the browser. The output charset determines what charset Active4D converts the response buffer *to* from Unicode.

You can set the output charset with the “output charset” config option in Active4D.ini. The possible options are the same as for the platform character set. You may also use the **set output charset** command.

The default output charset on Japanese language systems is “shift_jis”. The default output charset on Chinese language systems is “gb2312”. The default output charset on all other systems is “utf-8”.

Note: In previous versions of Active4D, the default platform charset for non-Japanese/Chinese systems, the default output charset was “iso-8859-1”. If your output is not in UTF-8 (which it should be!) and you did not set an output charset in Active4D.ini, you will have to set the output charset now.

Output Encoding

The output encoding determines how Active4D converts special characters to HTML character entities when text is written to the response buffer. Output encoding is performed *before* the output character set conversion, since the encoding tables are based on Unicode.

You can set the output encoding with the “output encoding” config option in Active4D.ini. You specify one or more bit flags to indicate which characters to encode. More than one flag can be specified by separating them by ‘+’ and any number of spaces. The bit flags are “none”, “quotes”, “tags”, “ampersand”, “extended”, “html” and “all” (without the quotes). Note that “extended” and “html” are synonymous.

You may also use the **set output encoding** command at runtime. For more information on output encoding, see “set output encoding” on page 305.

The default output encoding on Japanese or Chinese language systems is “none”. The default output encoding on all other systems is “html”.

Text is HTML encoded according to the following rules:

- If a character is ASCII, has a named entity, and the current encoding mode asks for that entity to be encoded, it is encoded.

- If a character is non-ASCII, the output charset is ISO-8859-1 or ISO-8859-15, the mode is “extended” or “all”, and the character has a named entity, it is encoded.
- Otherwise the character is passed through as is.

HTTP Request Decoding

When parsing an HTTP request, the headers are left as is. This is not a problem, since all headers except cookies will be in US ASCII and the character set is not an issue.

Query string parameters are automatically URL decoded and converted from UTF-8 to Unicode.

Posted form variables are automatically URL decoded and converted from the output character set to Unicode.

Informing the Browser of Your Output Character Set

You should always put the following tag in the header of HTML pages returned to a browser:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

The charset name at the end should match the charset name you use in the “output charset” config option. In the example above the output charset is “utf-8”.

This is critical for two reasons:

- The browser must know how to interpret non-ASCII characters in the page in order to correctly display them.
- When a form is posted, the browser encodes the form data in the character set of the page in which the form appears. There is unfortunately no mechanism within HTTP to communicate the character set of posted form data to the server. So Active4D must assume that the posted data is encoded in the output character set. Therefore it is critical that you use a Content-Type meta tag that matches your output character set.

Working with Files

Active4D cannot know the source platform of a file or its charset. It’s up to you to know the charset used by a file, then convert to the appropriate character set when saving to the database or writing to the browser.

Error Handling

Active4D goes to considerable lengths to catch errors and display meaningful error messages. If any errors occur during the execution of an Active4D program, execution is immediately aborted and the error handler takes over.

For complete information on error handling within Active4D, see Chapter 14, “Error Handling.”

Script Timeout

Despite our best intentions, sometimes our scripts may do bad things like going into infinite loops or waiting an inordinate amount of time for a shared resource.

Every script is given a set amount of time to execute. Before executing each line, Active4D checks to see if the timeout has been reached, and if so it generates an error and aborts execution.

You may set the script timeout with the “script timeout” config option in Active4D.ini. The value set with this option is the *minimum* script timeout in seconds and is the default value for all subsequent script executions.

The actual timeout can be set higher within Active4D with the **set script timeout** command. This command will affect the *next* execution of Active4D, not the one in which the command is used, and in no case can it be set lower than the minimum value set in Active4D.ini.

CHAPTER 8

Methods

As you build your web site with Active4D, you will undoubtedly come across chunks of code that can be reused in many different contexts, just as you would reuse code in 4D. Fortunately, Active4D has a powerful system for defining its own methods.

Defining Methods

The simplest way to define a method in Active4D is to declare it inline with the rest of your code, then reference it sometime later in the flow of execution.

Inline methods “live” only as long as the current invocation of Active4D. This means that you incur a small performance penalty each time the method is used, because it has to be parsed and stored in temporary memory before it can be used.

In general, you will only want to use inline methods during testing and debugging, because there is a much better way of defining and using methods: *libraries*. Libraries have many advantages over inline methods and allow you to group many methods together into one logical unit. They are covered completely in the next chapter, “Libraries”.

Method Declaration

Whether a method is defined inline or in a library, the syntax for declaring a method is the same:

```
method "<name>" {({&}$arg1{=expression}
                 {...{&}$argN{=expressionN}})}
  <statements>
end method
```

This may look complex, but it is actually quite simple. Keep reading.

Method Name

The method name must be a double-quoted literal string which follows the rules for 4D method names in terms of allowable characters. The maximum length of an Active4D method name is 255 characters. The double-quotes are necessary to allow you to include spaces as part of the method name, and to differentiate the method name from the **method** keyword.

Here is a simple Active4D method definition:

```
method "sayHello"  
    write("Hello world!")  
end method
```

Method Parameter Declaration

If a method takes parameters, the parameter list must follow the method name enclosed in parentheses. Unlike 4D, where method parameters are numbered, Active4D method parameters are declared by name, with each parameter becoming a local variable with that name within the body of the method. Parameter names must begin with '\$' and follow the rules for local variable names.

```
method "multiplyValues"($inValue1; $inValue2)  
    return ($inValue1 * $inValue2)  
end method
```

Note that there is currently no facility in Active4D for passing a variable number of parameters. You can simulate this technique by:

- Passing a *reference* to an array and accessing the elements of the array
- Passing a collection and accessing collection items by name
- Using *default parameters*, which are covered in the next section

Most of the time default parameters provide the best solution, so you should try to use them wherever possible.

Method Parameters

Parameters to Active4D methods differ from 4D method parameters in several important ways.

Parameter Type

There is no typing of parameters, as they are simply local variables with a variant type. This means you can pass values of different types in the same parameter, as long as the use of the parameter in the method would not cause any type incompatibilities. If you want to ensure a parameter is of a particular type, you would have to test its type within the method using the **Type** command.

Here is an example of how you can take advantage of the variant parameter typing:

```
method "writeMany"($inValue; $inHowMany)
  for ($i; 1; $inHowMany)
    writebr($inValue)
  end for
end method

writeMany("Hello world!"; 3)
writeMany(7.13; 2)

// The output:
Hello world!
Hello world!
Hello world!
7.13
7.13
```

Scope

Parameters passed to an Active4D method become a local variable within the scope of the method. Thus, like local variables in 4D, they have no existence outside of the method body.

```
method "paramTest"($inParam)
  $inParam:="bar" // The global $inParam is still "foo"
  writebr($inParam)
end method

$inParam:="foo"
writebr($inParam)
paramTest($inParam)
writebr($inParam)

// The output:
foo
bar
foo
```

Likewise, local variables defined outside of an Active4D method cannot *ordinarily* be referenced within the method.

```
method "localTest"
  write($local) // This generates an 'unknown identifier' error
end method

$local := "foobar"
localTest
```

Referencing “Global” Local Variables

Just as in 4D, it is best to pass whatever values a method needs as parameters. But what if you want to change the value of a local variable which was declared outside of a method? There are two ways to accomplish this: *pass by reference* and the **global** keyword.

Pass by reference allows you to effectively pass a pointer to a local variable. This is the preferred way of modifying locals external to the method, and is covered below. But at times it is preferable to reference an external local variable directly. For example, you may need to reference or modify many external variables within a method, and it would be too cumbersome to pass many parameters.

In these cases you may use the **global** keyword to declare a list of local variables that you want to make accessible to the method, as in this example:

```
method "writeNames"
  global($names)

  for($i;1;size of array($names))
    writebr($names{$i})
  end for
end method

array string(80;$ names; 0)
set array($names; "Tom"; "Dick"; "Harry")
writeNames
```

Note that the **global** keyword means just what it says: it looks for the named variables in the global scope. If the variable you want to reference was declared within another method, you must use **global** before declaring or first assigning that variable, which forces it into the global scope, like this:

```
method "outer"
  global($foo)
  $foo := "bar"
  inner
end method

method "inner"
  global($foo)
  write($foo) // writes "bar"
end method
```

Pass by Reference

Normally, parameters passed into Active4D methods are passed *by value*. In other words, the expression passed into the given parameter is evaluated and the constant value resulting from the expression is assigned to the parameter.

In addition to pass by value, Active4D also allows you to pass *by reference*. This powerful feature is activated by prefixing a parameter in the method argument list with an ampersand (&).

Reference parameters are essentially pointers to the entity that was passed into them. You can pass 4D pointers into Active4D methods, but there are two important ways in which reference parameters are unlike 4D pointers:

- You need not explicitly dereference the parameter to access the underlying value
- You can pass local variables by reference

Any entity that is assignable — including variables, fields, array elements, collection items and character references — may be passed by reference. You can even pass a character reference to an array element by reference!

Here is some examples of passing by reference:

```
method "paramTest"($inByValue; &$ioByReference)
  ++$inByValue
  ++$ioByReference
end method

$byValue := 7
$byReference := 7
paramTest($byValue; $byReference)

// At this point $byValue still = 7, but $byReference = 8

method "showArray"(&$inArray)
  for ($i;1;size of array($inArray))
    writebr($inArray{$i})
  end for
end method

array longint($longs; 0)
set array($longs; 7; 13; 27)
showArray($longs)

array text($names;0)
set array($names; "Tom"; "Dick"; "Harry")
showArray($names)
```

As you can see, the combination of 4D pointer support (mainly for tables and fields) and pass by reference allows you to write highly generic, reusable code within Active4D.

Default Parameters

In 4D, method parameters are numbered. Thus it is easy to pass a variable number of parameters. In Active4D, parameters are named, so there is another technique for passing variable number of named parameters. This technique is called *default parameters* and is another feature of Active4D borrowed from other languages.

To create a default parameter, append the parameter name with '=' and any expression which is valid at the point of method declaration. The expression is evaluated once when the method is first parsed, and the expression's value is stored with the method parameter. Within a default parameter expression, you can reference variables, methods, named constants, etc., as long as they are valid at the point of declaration.

Here is an example:

```
method "showArray"(&$inArray; $inDelimiter="<br />\n")
  writeln(join array($inArray; $inDelimiter))
end method

showArray($myArray)          // each element is on a separate line
showArray($myArray;" , ")  // elements are comma-delimited
```

In the first call to *ShowArray*, the *\$inDelimiter* parameter is not passed, so Active4D uses its default value of “
\n”. In the second call to *ShowArray*, the *\$inDelimiter* parameter was passed, so the passed value is used.

This is how you would have accomplished the same thing in 4D:

```
C_POINTER($1)
C_STRING(255;$2)

If (Count parameters < 2)
  $2:="<br />\n"
End if

`And so on
```

Returning Values

To return a value from a method, use the **return** keyword, like this:

```
method "concatStrings"($inFirst; $inSecond)
  return ($inFirst + $inSecond)
end method
```

The **return** keyword can also be used without a value, but if you return a value the expression must be enclosed in parentheses. Whether or not you return a value, the return keyword will immediately exit the method, no matter where in the method it occurs. This allows you to quickly exit a method when a termination condition is reached, much as the **break** keyword allows you to immediately exit a loop. It turns out this is extremely useful, as it saves you from having to set some kind of “success” flag which you test after the loop.

Here is an example:

```
method "findFirstOver"(&inArray; $inCompare)
  for ($i; 1; size of array($inArray))
    if ($inArray{$i} > $inCompare)
      return ($i) // The method terminates and returns $i
    end if
  end for

  // We only get here if the loop completes without a match
  return (-1)
end method
```


CHAPTER 9

Libraries

It is common for programmers to continually build up their own set of utility methods which they can call upon when needed. In 4D, these methods can be grouped together either by a name prefix or by placing them in a component.

In Active4D, you can group methods together into a named logical unit called a *library*. The library in effect serves as both the component and the name prefix.

Library Search Path

Active4D looks for libraries using the following search path:

- <Database folder>/Active4D (Standalone/Server)
- <User home>/Library/Application Support/4D/com.aparajita/Active4D/lib (macOS)
- <Disk>:\Users\<user>\AppData\Roaming\4D\com.aparajita\Active4D\lib (Windows 7 or Vista)
- <Disk>:\Documents and Settings\<user>\Application Data\4D\com.aparajita\lib (Windows XP)
- /Library/Application Support/4D/com.aparajita/Active4D/lib (macOS)
- <Disk>:\ProgramData\4D\com.aparajita\Active4D\lib (Windows 7 or Vista)
- <Disk>:\Documents and Settings\All Users\Application Data\4D\com.aparajita\Active4D\lib (Windows XP)
- Active4D.bundle/Contents/Resources/lib

Note: Active4D will follow aliases and symbol links on macOS and shortcuts on Windows.

Because Active4D will follow the entire search path, you can structure your libraries in a hierarchy, from the most specific to the most general, by placing them at various points in the search path.

In addition, you can specify a list of library folders to search in the config file. For more information, see the documentation for the “lib dirs” setting in the sample Active4D.ini config file.

Library Definition

Libraries are special files that can only contain the following entities:

- **import** keyword
- **define** keyword
- **library** keyword
- **end library** keyword
- Method definitions
- Comments

Any methods within a library file must be enclosed by a **library/end library** pair. The **library** keyword must be followed by a literal string which gives the name of the library. The library name must match the name of the library's definition file, minus the file extension. Libraries may not be nested.

Note: The names "Active4D" and "global" are reserved and may not be used as library names.

Here is an example library file called "mylib.a4l":

```
library "mylib"

define(kMyConstant; 7)

method "myMethod"($inMyParam)
  writebr($inMyParam)
end method

method "myConcat"($inText1; $inText2)
  return ($inText1 + $inText2)
end method

end library
```

Importing Libraries

To use the methods in a library, you must first *import* it. Importing can happen in several ways:

- Implicitly during startup if the library is a *circuit library*. For more information on circuit libraries, see "Circuit libraries and initializers" on page 550.
- By using the full *library.method* syntax.
- By using the **import** command.

The **import** command takes a library name as its argument — not a path, but simply the library name with no filename extension. The import process does the following:

- 1 Checks to see if the named library is already loaded into memory. If it has been nothing more is done.
- 2 If the library is not in memory, the library search path is followed to find a file called `<lib>.<ext>`, where `<lib>` is the library name in a full `<library>.<method>` call or the library name passed to the **import** command, and `<ext>` is the currently configured library extension. For information on the library search path, see “Library Search Path” on page 117.
- 3 If the library file is found, it is parsed. Active4D strips out all comments and empty lines, storing only the body of each method along with information about the method parameters. In addition, if any constants are defined in the library, they are stored as well.
- 4 If the library is parsed successfully, Active4D looks for a method in the library called `__load__`. If it exists, it is executed.
- 5 If no errors occurred up to this point, the library name is added to a list of imported libraries.
- 6 If any of the above steps fails, an error is generated and execution is aborted, unless `*` is passed as a second parameter to the **import** command, in which case a Boolean is returned to indicate the success of the import.

If you read the above steps closely, you will notice the following advantages that libraries have over inline methods:

- Libraries are only loaded and parsed once during the life of the server (unless you modify or explicitly unload them). Inline methods are loaded and parsed every time their page is executed.
- Comments within an inline method’s page must be parsed every time. Comments and empty lines in a library are not stored. Thus you may liberally comment library methods with no performance penalty.
- The methods in a library are scoped to the library’s name, whereas inline methods are global in scope. See the next section for more on this.

Import Configuration

Active4D follows a standard search path when attempting to locate a library during import. If you wish to put your libraries in a directory other than one in the standard search path, you may add the directory to the “lib dirs” path list in Active4D.ini.

By default the “lib dirs” path list is empty.

When importing a library, by default Active4D appends “.a4l” to the library name and searches for a file with that name. If you wish to use a different extension for libraries, you may change it by setting the “lib extension” option in Active4D.ini. The extension must be a dot followed by no more than five alphanumeric characters.

Import Errors

When parsing a library, if Active4D encounters a error in the syntax of the methods definitions, it generates an error message with the location and type of syntax error. If the library is imported as part of an executable request, a standard Active4D error

message is generated. If the library is being reloaded asynchronously during a refresh, the error message is logged.

Automatic Re-Import

Active4D periodically checks to see if the source file for each imported library has been modified. The interval between checks is set with the “refresh interval” option in Active4D.ini. If a library’s source file has been modified, it is re-imported automatically, so there is no need to quit and restart 4D to have the changes take effect.

Note: Once a library is imported you cannot move its source file to another location without explicitly flushing the library or restarting the Active4D server.

Library Namespace

Methods and constants defined in a library are stored in memory under the library’s name. The library name then becomes a *namespace* that encapsulates all of the methods defined within it. This allows you to define method names without worrying whether another developer has chosen to use the same method name. As long as they are in libraries with different names, you can access each method separately.

Name Resolution

When Active4D encounters a non-4D method name, it first searches all of the imported libraries for a method with that name. If there is more than one match, an error is generated and execution is aborted. If there is one and only one match that method is executed.

If two or more libraries are imported that have a method with the same name, you must *disambiguate* the method name by prepending the library name and a dot before the method name. Otherwise Active4D will stop and tell you that there is an ambiguous reference to a local method.

For example, suppose you have these two libraries:

```
library "foo"
  method "doSomething"
    return ("Hello from foo")
  end method
end library

library "bar"
  method "doSomething"
    return ("Hello from bar")
  end method
end library
```

To call the two “doSomething” methods, you must do the following:

```
writebr(foo.doSomething)    `Output is "Hello from foo"
writebr(bar.doSomething)    `Output is "Hello from bar"
```

Library scope

Methods and constants defined within a library may be referenced anywhere within that library without using the **library.entity** syntax. In fact, all entity names within a library take precedence over names external to the library.

This rule leads to the following type of problem:

```
method "showArray"(&$inArray)
  `Do something here
end method

`Imagine the following is within a separate library file
library "myUtils"

method "showArray"(&$inArray)
  `Do something here
end method

method "showArrayAndSomethingElse"(&$inArray, $inSomethingElse)
  showArray($inArray) `This will call myUtils.showArray()
  write($inSomethingElse)
end method

end library
```

In this example, the library method **showArrayAndSomethingElse** calls the method **showArray**. Because **showArray** is defined within the library, that version of the method takes precedence over the inline **showArray** method defined outside the library.

The “global” library

In the example above, what if you wanted to reference the inline **showArray** method within the library? Fortunately there is a way.

All methods that are defined inline, i.e. in global scope, are placed in a special library called “global” which is implicitly imported. The global library exists only as long as the current execution of the interpreter, and cannot be flushed programmatically. Any attempt to import a library with the name “global” will result in an error, as that name is reserved.

To reference a global method or constant within a library, you can (and probably should) use the form **global.method** or **global.constant**. This will ensure that the global version of the entity will be referenced.

Going back to the example above, to ensure the method **showArrayAndSomethingElse** calls the global method **showArray**, you would write it like this:

```
method "showArrayAndSomethingElse"(&$inArray, $inSomethingElse)
  global.showArray($inArray) `Calls the inline showArray()
  write($inSomethingElse)
end method
```

Private methods

If a library method becomes overly long and complex, you may wish to split it up into several smaller methods. Or there may be several methods within a library that share common code and you want to factor the common code into a separate method. However, you don't want these helper methods to be publicly accessible outside the library as they have no use by themselves.

You can make a method private to its library by prefixing the method name with an underscore("_"). Any attempt to call such a method from outside its library will result in an error.

For an example of this technique, you can examine the **newFromX** RowSet library methods, which use several private helper methods.

Library-private collections

In addition to private methods, libraries also are provided with a private collection in which to store library-private data. This collection is created automatically when a library is loaded during import and is cleared when a library is unloaded.

Note: Because the library-private collection is in effect global in scope, if you want to store another collection within it, it must be a global collection, and you must be sure to clear that collection in the library's `__unload__` method.

The library-private collection is accessed through the keyword **self**, which returns a handle to the collection. The **self** keyword may only be used within a library method. Attempting to use it outside of a library method will generate an error. For examples of how to use **self**, see "Library Initialization/Deinitialization".

Because library-private collections cannot be accessed from outside the library, you may want to provide a debugging method like the following to view **self**:

```
method "dumpData"
  a4d.debug.dump collection(self; \\  
                                current library name + ": self"; \\  
                                true)
end method
```

Library Initialization/Deinitialization

There are two special methods you may define within your libraries: `__load__` and `__unload__`. If present, the `__load__` method is executed when a library is loaded during import. If present, the `__unload__` method is executed when a library is unloaded during a flush (either from a direct command or as part of a refresh).

In conjunction with the **self** collection, these methods allow you to turn libraries into completely self-contained components.

For example, it is very common for a database to have lists of options that need to be presented as HTML lists on a page. In many cases these lists are static, or they come from the database but change very infrequently. In such cases you can cache the pre-built HTML option lists in a library that provides methods to return the option lists.

Here is what such a library might look like:

```
// Menu library for ORM project

library "orm.menus"

method "__load__"
  _loadAll
end method

method "_loadAll"
  _loadStates
  _loadCountries
  _loadStatuses
end method

method "_loadStates"
  self{"states"} := ""
  array string(2; $codes; 0)
  array string(31; $names; 0)
  split string(a4d.web.kUS_UnsortedStateCodes; ";"; $codes)
  split string(a4d.web.kUS_SortedStateNames; ";"; $names)

  for ($i; 1; size of array($codes))
    $opt := '<option value="$codes{$i}">$names{$i}</option>\n'
    self{"states"} += $opt
  end for
end method

// and so on for the other _load methods

method "getStateMenu"($inCode)
  $value := 'value="$inCode"'
  return (replace string(self{"states"}; $value; \\
    $value + " selected"))
end method
```

Note the use of a private `_loadAll` method, which actually does the loading. The reason for providing such a method is to enable the site administrator to reload the library's

cached data through the web by executing a library method from a web page. For example, we could add the following method to the above library:

```
method "reload"
  _loadAll
end method
```

Because we factored the loading logic into `_loadAll`, neither `__load__` nor `reload` will have to change if the loading logic changes.

Storing collections in library-private data

When storing collections in the library-private `self` collection, special care has to be taken, because `self` is private to the library but persistent in its lifetime. Therefore global collections must be stored in the `self` collection.

For example, let us assume that `orm.menus._loadStatuses` in the library example above fills a collection with contact status information. To implement this we would have to make the following changes:

```
method "__unload__"
  _unload
end method

method "reload"
  _unload
  _loadAll
end method

method "_loadStatuses"
  self{"status"} := new global collection
  all records([status])
  distinct values([status]type; $types)

  for ($i; 1; size of array($types))
    $type := $types{$i}
    query([status]; [status]type = $type)
    selection to array([status]code; self{"status"}{$type})
  end for
end method

method "_unload"
  clear collection(self{"status"})
end method
```

The most important change is the call to `_unload` in the `__unload__` and `reload` methods, which ensures the global collection `self{"status"}` is cleared.

Creating a Poor Man's Class

Libraries also allow you to create a class-like structure that can be used to make collections that act sort of like traditional objects. These "objects" act like objects

because they encapsulate private data, and you can call methods on these objects with an object-oriented style syntax.

To create a Poor Man's Class, follow these steps:

- 1 Create a library (we will use "Foo" in this example) that will act as the class.
- 2 Make one or more constructor methods (`newFoo`, `newFooFromBar`, etc.). In the process of construction you create a collection which becomes the "object".
- 3 Make sure the collection has an item called "`__class__`" whose name matches the library name. It is best to use the **current library name** command for this.
- 4 If you want the object's data to be private — which you should, as good object-oriented design dictates you should force data access to go through methods — prefix the key names with an underscore, which prevents them from being accessed outside of the object's library/class. So, for example, if the `newFromBar` constructor took a `bar` parameter, you would store it like this:

```
$object{"_bar"} := $inBar
```

- 5 Return the collection handle to the caller from the constructor, that is the object reference.
- 6 To define object methods, define methods with a first parameter called `$self`, which will receive the object reference (which is actually a collection handle). For example:

```
method "setBar"($self; $inBar)
  $self{"_bar"} := $inBar
end method

method "getBar"($self)
  return ($self{"_bar"})
end method
```

- 7 Given an object reference, you can call a method on that object using the dereference operator (this syntax is taken from C++/PHP), like this:

```
$foo := Foo.newFromBar("foobar")
write($foo->getBar)
$foo->setBar("barfoo")
```

When Active4D sees `->`, it transforms:

```
$object->method(params)
```

internally into:

```
$object{"__class__"}.method($object; $params)
```

So here is our complete Foo class:

```

library "Foo"

method "new"
  return (_init(""))
end method

method "newFromBar"($inBar)
  return (_init($inBar))
end method

method "_init"($inBar)
  $object := new collection("__class__"; current library name; \
                           "_bar"; $inBar)

  return ($object)
end method

method "setBar"($self; $inBar)
  $self{"_bar"} := $inBar
end method

method "getBar"($self)
  return ($self{"_bar"})
end method

end library

```

Limitations

There is a reason these are called “Poor Man’s Classes”. Real classes have inheritance, real constructors, destructors, and so on. In traditional terms, this technique is actually *object-based* programming, not *object-oriented* programming. Someday in the future we will have real classes and objects in Active4D, but for now this technique is quite sufficient to implement many object-oriented techniques into your applications.

For good examples of Poor Man’s Classes, take a look at the source code for RowSets and Breadcrumbs, both of which use this technique extensively.

CHAPTER 10

Event Handlers

In the course of executing a script, there are well-defined points at which a developer would like have some control.

Active4D recognizes special *event handler* methods which are executed before and after various “events.” To be activated, these event handler methods must be defined in a special library called “Active4D.<lib>”, which must reside in an Active4D directory. The “<lib>” extension must be whatever you have configured the library extension to be in the Active4D.ini file. By default the library extension is “.a4!”.

As with the config files, you may have multiple copies of the Active4D library in different directories in the search path. An Active4D library at the beginning of the search path will override one later in the search path.

Event Handler Methods

The event handler methods are outlined below. In addition to these, you may define and call other methods within the Active4D library just as you would with any library.

Note: You need not define an event handler if you are not going to use it.

Each handler is executed within a certain context which determines what data is accessible within the handler. For example, handlers that execute within the context of an HTTP request have access to all HTTP request and response collections which Active4D creates.

In order of execution (from startup to shutdown), the event handler methods are:

On Application Start

This handler is executed when 4D first starts up or after the Active4D server has been restarted. This handler is analogous to the *On Startup* database method in 4D.

The only Active4D collection you have access to in this handler is the *globals* collection. In addition, you can reliably read the following variables:

Variable	Description
<>A4D_HostAddr	IP address of host on which Active4D is running
<>A4D_HostPort	Port on which web server is listening

Variable	Description
<>A4D_HostType	Type of TCP/IP host ("4D", "NTK")
<>A4D_ClientIsWebServer	<i>True</i> if serving on 4D Remote

On Request

This handler is executed just before the Active4D HTTP server handles a request, before Active4D parses the request URL.

The handler is passed the path portion of the URL. The `_query` collection (or `_form` collection if "parameter mode" is set to "form variables" in Active4D.ini) is populated with the contents of the request's query (if any), and is set to read-write mode so that you can directly modify the query.

Within the handler, there are several actions you can take:

- **Leave the URL unchanged:** If you do not return any value or return an empty string, the URL is left unchanged.
- **Change the URL:** If you wish to change the URL, you may do so by returning a non-empty string. The URL must be non-URL encoded Unicode.
- **Modify the query:** If you wish to modify the query, you may directly modify the contents of the built in `_query` collection (or `_form` collection if "parameter mode" is set to "form variables" in Active4D.ini).
- **Reject the request:** You may refuse the request altogether by calling **set response status** with a status other than 200 (OK), such as 404 (Not Found) or 303 (See Other). If you set the response status to something other than 200, you do not need to return a result.
- **Redirect:** You may redirect to another URL by setting a response of 303 (See Other) or 301 (Moved Permanently) and setting a "Location" response header with the full target URL (including query), which should be url encoded UTF-8. If you do a redirect in this way, you do not need to return a result.

Here is what an *On Request* handler might look like:

```
// We want to change "/products/show?id=13"
// into "/index.a4d?action=show;id=13"

method "On Request"($inURL)
  // The URL is in the form /<circuit>/[<action>][?<query>].
  // If there is a dot in the filename, assume it is a
  // non-executable resource, and return the url as is.
  $circuit := directory of($inURL; *)
  $action := filename of ($inURL)

  if ( "." !~ $action)
    // If the url has only a circuit name with no trailing slash,
    // $circuit will be empty and $action has the circuit name.
    // In that case call the main action on the circuit.

    if (length($circuit) = 0)
      if (length($action) > 0)
        $circuit := $action
        $action := "main"
      end if
    else
      $circuit := substring($circuit; 2)

      if (length($action) = 0)
        $action := "main"
      end if
    end if

    if (length($circuit) > 0)
      _query{fusebox.conf.fuseaction} := '$circuit.$action'
    end if

    return ("/index.a4d")
  end if
end method
```

The *request info*, *request cookies*, *response headers* and *response cookies* collections are accessible within this handler. This allows you to check things like the host, set cookies, etc.

On Authenticate

When Active4D determines that the current request is in a protected realm, if this event handler is defined it is invoked before the *On Session Start* and *On Execute Start* event handlers.

Here is what a sample *On Authenticate* handler might look like:

```
method "On Authenticate"
  if (auth user = "")
    authenticate
  else
    query([security];[security]realm = current realm;*)
    query([security];&[security]username = auth user;*)
    query([security];&[security]password = auth password)

    case of
      : (records in selection([security]) = 0)
        authenticate
      : (not(identical strings([security]password; \\\
                                auth password)))
        authenticate
    end case
  end if
end method
```

In this example we are using a table that defines all of the users and passwords for each realm. If we find a match, we check the passwords to make sure the capitalization is exactly the same. If there is no match, we authenticate again.

Be sure to pass along an authentication failure message with the **authenticate** command, either by writing directly to the response buffer, by including another file, or by utilizing the standard HTTP error handling mechanism as described in “HTTP Error Handling” on page 63. When the **authenticate** command returns, the HTTP status code is 401.

Within this handler you may access all of the Active4D collections.

On Session Start

This handler is executed when a new user (one with no current session) makes a successful HTTP request (one with a result code of 200 OK), before the *On Execute Start* handler.

Note: Whether or not you set any session items in *On Session Start*, if this event handler is defined a new session will *always* be created for *each* new visitor to your site. Therefore you should *only* define this method if your intention is to track each new visitor through a session.

For example, here is an *On Session Start* handler that initializes three items in a session:

```
method "On Session Start"
  session{"start"} := 1
  session{"recsPerPage"} := 10
  session{"sortAscending"} := true
end method
```

The first time a user makes a request from your site, these three session items will be set. For more information on sessions, see “Sessions” on page 343.

If you want to initialize new sessions only when a certain event occurs, such as a successful user login, do not use this handler. Rather, initialize your session at the point at which you determine one is needed.

You can use the **requested url** command to determine which part of your web site was accessed, thus determining how to initialize the session. You may also use the **redirect** command to force all new sessions to go to a login page, for example.

For the redirect trick to work, if you are using session cookies the user must have cookies on, otherwise you will end up in an endless loop. To prevent this, in the redirect page you must check for the presence of the session cookie, and if it is not there you must then redirect to a *static* HTML page (not an executable page!) which tells the user they must turn cookies on.

Within this handler you may access all of the Active4D collections.

On Execute Start

This handler is executed before Active4D begins execution. Typical uses for this handler would include dumping some debugging information, or initializing timing information.

Because this handler is executed before the requested file is parsed, any HTML you write to the response buffer will appear *before* the opening `<html>` tag. It is possible that some browsers may not like this, although both Internet Explorer and Netscape seem to handle it without problems.

Within this handler you may access all of the Active4D collections.

On Execute End

This handler is executed after Active4D completes execution, unless the **redirect** command was called.

Within this handler you may access all of the Active4D collections.

On Session End

This handler is executed when a session goes to heaven (if it has been good), either because it timed out, it was expired with the **abandon session** command, the Active4D server has been restarted, or because the server is shutting down.

Note: On Session End is not invoked if you are using a custom session handler. For more information, see “Session Handlers” on page 348.

Typically you would use this handler to clean up data that was stored during the course of the user’s session. For example, if the user uploaded a file and you stored the path to the file in the session, when the session times out you would use this handler to delete the file.

Because this handler is executed asynchronously at idle time, the only Active4D collection you have access to is the globals collection, as there is no request context. However, the about-to-be-purged session is made current and you can access all of its data one last time before kissing it goodbye.

There are a couple of important points to note in regards to this handler:

- If you wish to identify a session persistently, always use **session internal id**, as the result of **session id** is undefined in this handler.
- There is no deterministic way of knowing when this handler will run. The only thing you know for sure is that it will run sometime after a session expires.
- If multiple sessions have expired when a session purge cycle begins, this handler will be run once for each session. However, there is no deterministic way of knowing the order in which the sessions will be handled.

On Application End

This handler is executed just before the server shuts down, either when 4D is shutdown or when the Active4D server is restarted. This handler is analogous to the *On Exit* database method in 4D.

The only Active4D collection you have access to in this handler is the globals collection.

Modifying the Active4D Library

Because the Active4D library has event handlers that are run only at application start and application shutdown, it is not automatically reloaded when it is modified.

If you want to change the methods in the Active4D library while 4D is running, delegate the Active4D methods to another library. For example, you can create another library called “_active4d.a4!”, move your event handling code there, and do this in Active4D.a4!:

```
method "On Request"($inURL)
    return (_active4d.onRequest($inURL))
end method

method "On Authenticate"
    _active4d.onAuthenticate
end method

// and so on
```

This technique allows you to make modifications to the event handling code without affecting the Active4D library itself. Because the “_active4d” library is a normal library, when you modify the code in that library, the library will automatically be reloaded.

CHAPTER 11

Command Reference

Active4D implements over 460 commands that provide unparalleled power and simplicity to web site programming. Of these commands, over 170 are 4D commands. The remaining 300 commands are specific to Active4D.

4D Commands

Following is a list of the 4D commands implemented by Active4D. Unless indicated by the formats noted below, they take the same parameters and work exactly as they do in 4D.

- Commands in **bold** have been enhanced by Active4D.
- Commands in *italics* have a limitation relative to the 4D v11+ version.

Any commands which were added or extended in 4D v11+ are implemented in their extended form in Active4D.

Note: Unlike 4D, the 4D commands used by Active4D are always in English. If you are using a foreign language version of 4D, please be aware that you will have to use English commands and named constants in Active4D.

Using a Default Table

You cannot use a default table in Active4D. All commands that *may* take a table in 4D *must* be given one in Active4D.

4D Commands Supported by Active4D

Abs	Command name	Find index key
Add to date	COPY ARRAY	FIRST RECORD
ADD TO SET	COPY DOCUMENT	FOLDER LIST
ALL RECORDS	COPY NAMED SELECTION	Get character code
Append document	COPY SET	Get document position
APPEND TO ARRAY	Count fields	Get document size
ARRAY BLOB	Count in array	GET FIELD PROPERTIES
ARRAY BOOLEAN	Count tables	Get indexed string
ARRAY DATE	Create document	Get last field number
ARRAY INTEGER	CREATE EMPTY SET	Get last table number
ARRAY LONGINT	CREATE FOLDER	GET PICTURE FROM LIBRARY
ARRAY PICTURE	CREATE RECORD	Get pointer
ARRAY POINTER	CREATE SELECTION FROM ARRAY	GOTO RECORD
ARRAY REAL	CREATE SET	GOTO SELECTED RECORD
ARRAY STRING	CREATE SET FROM ARRAY	INSERT ELEMENT
ARRAY TEXT	Current date	INSERT IN ARRAY
ARRAY TIME	Current method name	Insert string
Ascii	Current process	Int
AUTOMATIC RELATIONS	Current time	INTERSECTION
Average	CUT NAMED SELECTION	Is field number valid
Before selection	Date	Is in set
BLOB size	Day number	Is table number valid
BLOB TO DOCUMENT	Day of	ISO to Mac (<i>deprecated</i>)
BLOB to longint	Dec	LAST RECORD
BLOB to text	DELAY PROCESS	Length
C_BLOB	DELETE DOCUMENT	LIST TO ARRAY
C_BOOLEAN	DELETE ELEMENT	LOAD RECORD
C_DATE	DELETE FOLDER	Locked
C_LONGINT	DELETE FROM ARRAY	LONGINT TO BLOB
C_PICTURE	DELETE RECORD	Lowercase
C_POINTER	DELETE SELECTION	Max
C_REAL	Delete string	Milliseconds
C_STRING	DIFFERENCE	Min
C_TEXT	DISTINCT VALUES	Month of
C_TIME	DOCUMENT LIST	MOVE DOCUMENT
CANCEL TRANSACTION	DOCUMENT TO BLOB	NEXT RECORD
Char	End selection	Nil
Character code	EXECUTE	Not
CLEAR NAMED SELECTION	False	Num
CLEAR SEMAPHORE	Field	ONE RECORD SELECT
CLEAR SET	Field name	Open document
CLEAR VARIABLE	Find in array	ORDER BY
CLOSE DOCUMENT	Find in field	ORDER BY FORMULA

4D Commands Supported by Active4D (cont.)

PICTURE PROPERTIES	SET QUERY LIMIT
Picture size	Size of array
Position	SLEEP
PREVIOUS RECORD	SORT ARRAY
QUERY	START TRANSACTION
QUERY BY FORMULA	String
QUERY SELECTION	STRING LIST TO ARRAY
QUERY SELECTION BY FORMULA	Structure file
QUERY SELECTION WITH ARRAY	Substring
QUERY WITH ARRAY	Sum
Random	Table
READ ONLY	Table name
Read only state	Test path name
READ PICTURE FILE	Test semaphore
READ WRITE	TEXT TO BLOB
RECEIVE PACKET	Tickcount
Record number	Time
Records in selection	Time string
Records in set	Trunc
Records in table	True
REDUCE SELECTION	Type
RELATE MANY	Undefined
RELATE MANY SELECTION	UNION
RELATE ONE	UNLOAD RECORD
RELATE ONE SELECTION	Uppercase
REMOVE FROM SET	USE NAMED SELECTION
Replace string	USE SET
Resolve path	VALIDATE TRANSACTION
RESOLVE POINTER	WRITE PICTURE FILE
Round	Year of
SAVE RECORD	
SCAN INDEX	
Selected record number	
SELECTION RANGE TO ARRAY	
SELECTION TO ARRAY	
Semaphore	
SEND PACKET	
Sequence number	
SET AUTOMATIC RELATIONS	
SET BLOB SIZE	
SET DEFAULT CENTURY	
SET DOCUMENT POSITION	
<i>SET QUERY DESTINATION</i>	

Active4D Commands

Active4D implements almost 300 of its own commands. Most are focused on web programming. Some are additions to the 4D language that we have always wanted. The bottom line is this: if you learn these new commands, you will be far more productive. So please take the time to learn them!

Active4D Commands

_form	count collection items	format string
_query	count form variables	full requested url
_request	count globals	get auto relations
abandon response cookie	count query params	get cache control
abandon session	count request cookies	get call chain
add datetime to json	count request infos	get collection
add element	count response cookies	get collection array
add function to json	count response headers	get collection array size
add rowset to json	count session items	get collection item
add selection to json	count uploads	get collection item count
add to json	current file	get collection keys
add to timestamp	current library name	get content charset
append to array	current line number	get content type
auth password	current path	get current script timeout
auth type	current platform	get error page
auth user	current realm	get error status
authenticate	day of year	get expires
auto relate	deep clear collection	get expires date
base64 decode	deep copy collection	get field numbers
base64 encode	default directory	get field pointer
blob to collection	define	get form variable
blob to session	defined	get form variable choices
blowfish decrypt	delete collection item	get form variable count
blowfish encrypt	delete global	get form variables
buffer size (<i>deprecated</i>)	delete response cookie	get global
build query string	delete response header	get global array
call 4d method	delete session item	get global array size
call method	directory exists	get global item
capitalize	directory of	get global keys
cell	directory separator	get http error page
choose	enclose	get item array
clear array	end json array	get item key
clear buffer (<i>deprecated</i>)	end json object	get item type
clear collection	end save output	get item value
clear response buffer	execute in 4d	get library list
collection	extension of	get license info
collection has	file exists	get local
collection to blob	filename of	get log level
compare strings	fill array	get output charset
concat	first not of	get output encoding
configuration	first of	get platform charset
copy collection	form variables	get query param
copy upload	form variables has	get query param choices

Active4D Commands (cont.)

get query param count
get query params
get request cookie
get request cookies
get request info
get request infos
get request value
get response buffer
get response cookie
get response cookie domain
get response cookie expires
get response cookie path
get response cookies
get response header
get response headers
get response status
get root
get script timeout
get session
get session array
get session array size
get session item
get session names
get session stats
get session timeout
get throw code
get throw message
get time remaining
get timestamp datetime
get upload content type
get upload encoding
get upload extension
get upload remote filename
get upload size
get utc delta
get version
global
globals
globals has
hide session field
html encode
identical strings
import
in error
include
include into
insert into array
interpolate string
is a collection
is an iterator
is array
join array
join paths
json encode
json to text
last not of
last of
left trim
library list
load collection
local datetime to utc
local time to utc
local variables
lock globals
log message
longint to time
mac to html
max of
md5 sum
merge collections
method exists
min of
more items
multisort arrays
multisort named arrays
native to url path
new collection
new global collection
new json
new local collection
next item
nil pointer
param text
parameter mode
parse json
query params
query params has
random between
redirect
regex callback replace
regex find in array
regex find all in array
regex match
regex match all
regex quote pattern
regex replace
regex split
request cookies
request info
request query
requested url
require
resize array
response buffer size
response cookies
response headers
right trim
save collection
save output
save upload to field
session
session has
session id
session internal id
session local
session query
session to blob
set array
set cache control
set collection
set collection array
set content charset (*deprecated*)
set content type
set current script timeout
set error page
set expires
set expires date
set global
set global array

Active4D Commands (cont.)

set http error page
set local
set log level
set output charset
set output encoding
set platform charset
set response buffer
set response cookie
set response cookie domain
set response cookie expires
set response cookie path
set response header
set response status
set script timeout
set session
set session array
set session timeout
slice string
split path
split string
start json array
start json object
throw
time to longint
timestamp
timestamp date
timestamp day
timestamp difference
timestamp hour
timestamp millisecond
timestamp minute
timestamp month
timestamp second
timestamp string
timestamp time
timestamp year
trim
type descriptor
unlock globals
upload to blob
url decode
url decode path
url decode query
url encode
url encode path
url encode query
url to native path
utc to local datetime
utc to local time
variable name
week of year
write
write blob
write gif
write jpeg
write jpg
write json
write jsonp
write raw
write to console
writebr
writeln
writep

Command Syntax

The commands in this chapter are listed with the same basic format that the 4D documentation uses, with a few small differences:

- Commands unique to Active4D are all lowercase to distinguish them from 4D commands which have been implemented by Active4D.
- Parameters have a prefix to indicate what happens to them within the body of the method. The prefix “in” means the value of the parameter is read but not written. The prefix “io” means the value is both read and written. The prefix “out” means the value is written, replacing any existing value.

Unicode and Charsets

Because Unicode is used throughout Active4D, there are times when a command needs to convert text to or from Unicode. In those cases an *inCharset* parameter is provided, which should be a valid IANA character set name (or alias). For a list of IANA character set names and aliases, see:

<http://demo.icu-project.org/icu-bin/convexp?s=IANA&s=ALL>

Arrays

Active4D adds several commands and an extended syntax which make working with arrays *much* easier. It will pay many times over for you to learn and use these commands.

{} (appending index)

version 4.0

```
<array>{}
```

Discussion

Active4D adds a new syntax for appending elements to an array. If you use an empty index with an array, a new element is appended to the array and the index is set to the newly appended element.

This allows you to add items to an array in a very simple way, like this:

```
array longint($longs; 0)
$longs{} := 7 // same as append to array($longs; 7)
$longs{} := 13 // same as append to array($longs; 13)

// $longs now contains 2 elements, 7 and 13
```

{-<index>} (from end index)

version 4.5

```
<array>{-<index>}
```

Discussion

Active4D adds a new syntax for referencing elements relative to the end of an array.

By using a negative index, you index array elements from the end of the array, with -1 being the last element and *-Size of array* being the first element.

For example, to reference the last element of an array, you can simply do this:

```
$myArray{-1} := 7

// old way, which one is easier?
$myArray{size of array($myArray)} := 7
```

add element

version 1

```
add element(ioArray {; inHowMany})
```

Parameter	Type	Description
ioArray	Array	→ The array to which you want to append elements
inHowMany	Number	→ How many elements to append

Discussion

This command appends one or more empty elements to *ioArray*. If *inHowMany* is omitted, one element is appended to the end of *ioArray*. The element appended to the array is initialized to the default value for the array's type. This command is essentially shorthand for the following standard 4D statement:

```
insert element($ioArray; size of array($ioArray) + 1; $inHowMany)
```

append to array

version 1 (modified v2)

```
append to array(ioArray; inValue {; inValueN})
```

Parameter	Type	Description
ioArray	Array	→ The array to which you want to append elements
inValue	<any>	→ Elements to append

Discussion

This command appends one or more values to the existing contents of *ioArray*. If a value is not assignment compatible with *ioArray*, an error is generated and execution is aborted.

Unlike the 4D version of this command, in Active4D you can append multiple values at once, like this:

```
append to array($array; "one"; "two"; "three")
```

Note: If you want to unconditionally set the contents of an array when it is declared, it is easier to use the extended array declaration syntax. If you want to reset the contents of an array after it has been created, as opposed to appending, you might want to use the **set array** command.

ARRAY <type>**4D version 3
version 1 (modified v6.1r6)**

ARRAY <type>({inStringWidth; } outArray; *, inValue {; inValueN})

Parameter	Type	Description
inStringWidth	Number	→ Width for fixed string elements
outArray	Array	→ The array to be created/resized
*	*	→ Indicates inline value setting
inValue	<any>	→ Elements to set

Discussion

This command allows you to declare or resize an array and define its elements in one statement.

If *type* is a **STRING** or **TEXT**, the values are automatically converted to the appropriate type with the default string conversion. For all other types, if a value is not assignment compatible with *type*, an error is generated and execution is aborted.

Note: As with 4D in Unicode mode, **ARRAY STRING** actually creates **ARRAY TEXT**.

If you are running Active4D in 4D v14 or later, the **ARRAY BLOB** and **ARRAY TIME** commands may be used to create BLOB and Time arrays. Except where noted, you may use an element of a BLOB or Time array anywhere a command is documented as taking or returning a BLOB or Time.

Note: If you attempt to use these commands with an earlier version of 4D, an error is generated and execution is aborted.

Examples

```
// 4D way
array text($items; 3)
$items{1} := "one"
$items{2} := string(current date)
$items{3} := string(?07:27:13?)

// easier Active4D way
array text($items; 0)
set array($items; "one"; string(!04/13/64!); string(?07:27:13?))

// easier way
array text($items; *; "one"; !04/13/64!; ?07:27:13?)
```

clear array**version 2**

clear array(ioArray {; ...ioArrayN})

Parameter	Type	Description
ioArray	Array	→ Array to clear

Discussion

This command resizes one or more arrays to zero elements.

COPY ARRAY**(modified 4D) version 3
version 4.0**

COPY ARRAY(inSourceArray; ioDestArray {; inStart; inEnd})

Parameter	Type	Description
inSourceArray	Array	→ Array from which to copy
ioDestArray	Array	→ Array to receive copied elements
inStart	Number	→ Starting element to copy
inEnd	Number	→ Ending element to copy

Discussion

If *inStart* and *inEnd* are not passed, this command works exactly like it does in 4D.

If *inStart* and *inEnd* are passed, *ioDestArray* will receive a copy of that subrange of elements.

If *inStart* is zero, elements are copied starting at the zero element of *ioDestArray*, otherwise elements are copied starting at the first element of *ioDestArray*.

Count in array

(modified 4D) v5Count in array(*inArray*; *inValue* {; *inStart*{}) → Number

Parameter	Type	Description
<i>inArray</i>	Array	→ Array to search
<i>inValue</i>	Array	→ Value to search for
<i>inStart</i>	Number	→ Element at which to start searching
Result	Number	← Occurrences of <i>inValue</i>

Discussion

This command differs from the 4D version of this command in that you can pass *inStart* to specify where to start searching for *inValue*.

If *inStart* is not passed, it defaults to 1. If *inStart* is less than zero, it is pinned to zero. If it is greater than the size of *inArray*, the result will be zero.

fill array

version 4.0fill array(*ioArray*; *inStart* {; *inEnd* {; *inStep* {; *inArrayStart*{}}})

Parameter	Type	Description
<i>ioArray</i>	Longint/Real array	→ The array to fill
<i>inStart</i>	Number	→ Starting number in sequence or size of sequence
<i>inEnd</i>	Number	→ Ending number in sequence
<i>inStep</i>	Number	→ Difference between steps in sequence
<i>inArrayStart</i>	Number	→ Where to start filling array

Discussion

This command fills an array with a sequence of values. In its simplest form, you can just pass an array and an integral number. In that case the array will be filled with a sequence of values from 1 to *inStart* inclusive, starting at element 1 of *ioArray*.

If *ioArray* does not exist when this command is called, it is created as a *Real* array.

If both *inStart* and *inEnd* are passed, the sequence will start at *inStart* and increment by 1 until **Abs(step) > Abs(inEnd)**. If *inStep* is passed, the sequence will start at *inStart* and increment by *inStep* until **Abs(step) > Abs(inEnd)**. All of these parameters may be non-integral.

If *inArrayStart* is passed, the sequence begins at that element. If *inArrayStart* > **Size of array(ioArray)**, the array is extended accordingly.

Examples

```
// fill an array with numbers 1-100
fill array($array; 100)

// fill an array with numbers 10 to -10
fill array($array; 10; -10; -1)

// fill an array with numbers from 1 to 2.5 by .5
fill array($array; 1; 1; 2.5; .5)
```

insert into array**version 4.0**

insert into array(*ioArray*; *inWhere*; *inValue* {; *inValueN*})

Parameter	Type	Description
<i>ioArray</i>	Array	→ Array into which to insert
<i>inWhere</i>	Number	→ Element before which values are inserted
<i>inValue</i> .. <i>inValueN</i>	<any>	→ Values to insert

Discussion

This command inserts one or more values into the existing contents of *ioArray*. The first inserted value will be element *inWhere*. If *inWhere* > **Size of array**(*ioArray*), the values are appended.

If a value is not assignment compatible with *ioArray*, an error is generated and execution is aborted.

is array

version 2

is array(inType) → Boolean

Parameter	Type	Description
inType	Longint	→ The type to test
Result	Boolean	← True if given type is an array type

Discussion

This command returns *true* if the given variable type is an array type. It is shorthand for the following test:

```
// 4D way
$isArray := ((type($var) >= Array 2D) & \\  
            (type($var) <= Boolean array))

// Active4D way
$isArray := is array(type($var))
```

join array

version 2

join array(inArray; inSeparator {; inStart {; inPrefixNum {; inQuoteText{}}) → Text

Parameter	Type	Description
inArray	Array	→ The array to join
inSeparator	Text	→ The text to insert between elements
inStart	Number	→ The element to begin joining from
inPrefixNum	Boolean	→ True to prefix the element number
inQuoteText	Boolean	→ True to quote-enclose elements of Text or String arrays
Result	Text	← Concatenation of array elements

Discussion

This command joins the elements of *inArray* together into a single string. Non-textual array elements are automatically converted to text.

If *inStart* is not specified, it defaults to 1.

If *inPrefixNum* is not specified, it defaults to *False*. If it is specified and *True*, each element is prefix by "{#} ", where # is the element number.

If *inQuoteText* is not specified, it defaults to *False*. If it is specified and *True*, elements of text or string arrays are surrounded by double quotes.

Here are some examples:

```
array longint($longs;0)
set array($longs; 7; 13; 27)
writebr(join array($longs; ", "))

array text($nums;0)
set array($nums; "one"; "two"; "three")
writebr(join array($nums; "<br />"; 1; true; true))

// Here is the output in the browser
7, 13, 27
{1} "one"
{2} "two"
{3} "three"
```

This command is especially useful for writing the contents of an array to the Active4D debugging console. Use this form:

```
write to console(join array($array; "\r"; 1; true; true))
```

multisort arrays

version 3.0

```
multisort arrays(inArray1; inDirection1 { ...inArrayN; inDirectionN})
```

Parameter	Type	Description
inArray	Array	→ The array to sort
inDirection	<>= or Text	→ The sort direction

Discussion

This command performs a multilevel sort on the elements of *inArray1* through *inArrayN*. The direction of the sort for each array is specified by the *inDirection* argument following the array:

Character	Direction
>	Ascending
<	Descending
=	Don't care, follow array to left

You may sort any array type except for picture arrays and pointer arrays. The arrays may be local, process, or interprocess variables.

The sort direction may be one of the operators '>', '<' and '=', or may be any text expression which resolves to one of those characters. This allows you to programmatically set the direction of the sort.

multisort named arrays

version 3.0

multisort named arrays(inArrayName1; inDirection1 {; ...inArrayNameN; inDirectionN})

Parameter	Type	Description
inArrayName	Text	→ The name of an array to sort
inDirection	<>= or Text	→ The sort direction

Discussion

This command is identical to *multisort arrays*, except that instead of passing direct array references, you pass text expressions which resolve to the names of arrays. This allows you to programmatically determine both the order and direction of the sort.

The arrays names should begin with '\$' to indicate a local array, '<>' to indicate an interprocess array, and no prefix to indicate a process array.

resize array

version 3.0

resize array(ioArray; inSize)

Parameter	Type	Description
ioArray	Array	↔ The name of an array to sort
inSize	Number	→ The new size of the array

Discussion

This command resizes *ioArray* to the given size. If *inSize* is less than zero, the array will be resized to zero.

SELECTION/SELECTION RANGE TO ARRAY

**(modified 4D) version 2
modified version 4.0**

SELECTION TO ARRAY
SELECTION RANGE TO ARRAY

Discussion

These commands have been enhanced in that the array arguments may be any valid array reference, including collection items which do not yet exist. This allows you to load data directly into collections (including built in collections like *session*), like this:

```
selection to array([ingredients]id; session{"ids"}; \\  
                  [ingredients]name; session{"names"}; \\  
                  [vendors]name; session{"vendors"})
```

set array**version 2**

```
set array(ioArray; inValue {; inValueN})
```

Parameter	Type	Description
ioArray	Array	→ The array which you want to set to the given elements
inValue	<any>	→ Elements to set

Discussion

This command replaces the existing contents of *ioArray* with one or more values. If a value is not assignment compatible with *ioArray*, an error is generated and execution is aborted.

This command is the fastest way to initialize an array to a known set of values. For example:

```
array longint($primes;0)
set array($primes; 2; 3; 5; 7; 11; 13)
```

BLOBS

The BLOB commands supported by Active4D function exactly as they do in 4D. Active4D adds two named constants for use with the **LONGINT TO BLOB** and **BLOB to longint** commands:

- **Intel byte ordering:** This is the equivalent of the 4D named constant *PC byte ordering*.
- **PPC byte ordering:** This is the equivalent of the 4D named constant *Macintosh byte ordering*.

You should prefer the Active4D equivalents over the 4D constants because they avoid the confusion arising from the fact that all new Macintoshes have used so-called “PC byte ordering” for several years, ever since the move to Intel processors.

Collections

The collection commands allow you to create, manipulate, examine and destroy your own local (temporary) or global (persistent) collections within your scripts.

For more information on collections, see “Collections” on page 99.

Note: Many of the collection commands are no longer necessary as they have been replaced with the extended indexing syntax, which is much easier to use. They have been retained for backwards compatibility. For more information, see “Referencing Collection Values” on page 101.

Note: In previous versions of Active4D, collection keys were kept in alphabetical order, although officially no order was guaranteed. In v6 the keys are kept in no predictable order, so if you have code that relies on alphabetical keys, you will have to get the collection keys, sort them, and then use the sorted keys to access the collection items.

collection**version 2**

collection(inHandle) → Longint

Parameter	Type	Description
inHandle	Longint	→ Collection handle
Result	Longint	← Iterator reference

Discussion

Given a collection, this command returns an iterator to the first item in the collection.

For more information on iterators, see “Iterators” on page 214.

new collection**version 2**
modified version 3.0

new collection>(* {; inKey; inValue {; inKeyN; inValueN}}) → Longint

Parameter	Type	Description
*	*	→ Pass to create a global collection
inKey	Text	→ Item key
inValue	<any>	→ Item value
Result	Longint	← Collection handle

Discussion

This command creates a new local or global collection and returns a handle to the collection. You then use this handle with the other collection commands.

If no * is passed, this command is exactly equivalent to **new local collection**. If * is passed, this command is exactly equivalent to **new global collection**.

You may also initialize the collection with key/value pairs by passing pairs of parameters. If an array is passed as the value, it is stored in its entirety in the item.

For example, this code would create a local collection and initialize it with two items:

```
$person := new collection("name"; [People]Name; \\  
                        "age"; [People]Age)
```

new local collection

version 2

```
new local collection{(inKey; inValue {; inKeyN; inValueN})} → Longint
```

Parameter	Type	Description
inKey	Text	→ Item key
inValue	<any>	→ Item value
Result	Longint	← Collection handle

Discussion

This command creates a new local collection and returns a handle to the collection. You then use this handle with the other collection commands. This command is exactly equivalent to **new collection**.

A collection created with this command is automatically cleared by Active4D when the script finishes execution.

Warning: You should always assign a local collection handle to a local variable or local collection item in the same scope.

new global collection

version 2
modified version 3.0

```
new global collection{(inKey; inValue {; inKeyN; inValueN})} → Longint
```

Parameter	Type	Description
inKey	Text	→ Item key
inValue	<any>	→ Item value
Result	Longint	← Collection handle

Discussion

This command creates a new global collection and returns a handle to the collection. You then use this handle with the other collection commands. This command is exactly equivalent to **new collection(*)**.

A collection created with this command remains in memory until the server shuts down or until it is deleted with **clear collection** or **deep clear collection**.

Warning: Be sure to store the handle in a place where you can retrieve it later, such as in the globals collection.

collection to blob

version 3.0
modified v5.0

collection to blob(inRef {; ioBlob}) → BLOB | <none>

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
ioBlob	BLOB	↔ BLOB to append data to
Result	BLOB	← Serialized collection data

Discussion

This command serializes the data in the collection referenced by *inRef*. If *ioBlob* is passed, the serialized data is appended to *ioBlob* and nothing is returned. If *ioBlob* is not passed, the serialized data is returned as a new BLOB. In either case, you can store the BLOB somewhere for later restoral via **blob to collection**.

To recursively serialize a collection, use the library method **a4d.utils.collectionToBlob**.

Warning: Do not attempt to serialize a v6 collection and then load it with a previous version of Active4D.

Note: If the collection contains BLOB or Time arrays, it must be deserialized in 4D v14 or later. Attempting to deserialize in earlier versions will generate an error.

blob to collection

version 3.0
modified v5.0

blob to collection(inBLOB {; ioOffset }; *) → Longint

Parameter	Type	Description
inBLOB	BLOB	→ BLOB with serialized collection data
ioOffset	Number	↔ Offset within BLOB to get data
*	*	→ Create a global collection
Result	Longint	← Collection handle

Discussion

This command creates a new collection from the serialized collection data contained in *inBLOB*. If *ioOffset* is passed, the serialized data must begin at that byte offset within *inBLOB*. After the collection is successfully deserialized, *ioOffset* will point to the first byte beyond the serialized data.

If *inBLOB* was not created with **collection to blob**, an error will be generated and execution will be aborted.

If the optional * parameter is given, the new collection is a global collection, otherwise the collection is local and will be cleared when the current script is finished executing..

Note: Active4D v6 will deserialize collections serialized with previous versions.

save collection

version 4.0

save collection(inRef; inPath)

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
inPath	Text	→ File path

Discussion

This command directly saves a serialized collection. It is effectively the same as:

```
$blob := collection to blob($inRef)
blob to document($inPath; $blob)
```

See **collection to blob** for notes and warnings about serializing a collection.

load collection

version 4.0

load collection(inPath) → Longint

Parameter	Type	Description
inPath	Text	→ File path
Result	Longint	← Collection handle

Discussion

This command directly loads a serialized collection. It is effectively the same as:

```
document to blob($inPath; $blob)
$collection := blob to collection($blob)
```

Note: Active4D v6 will deserialize collections serialized with previous versions.

copy collection

version 3.0
modified version 4.0

copy collection(inRef {; *}) → Longint

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
*	*	→ Create a global collection
Result	Longint	← Collection handle

Discussion

This command makes a shallow copy of the collection referenced by *inRef* and returns a handle to the copy. You can then use this handle with the other collection commands.

If the optional * parameter is given, the new collection is a global collection, otherwise the collection is local and will be cleared when the current script is finished executing.

Because this command makes a shallow copy of *inRef*, collection references embedded within *inRef* will not be copied, and the original will still be referred to in the copy.

To make a deep (recursive) copy, use the **deep copy collection** command.

deep copy collection

version 4.0

deep copy collection(inRef {; *}) → Longint

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
*	*	→ Create a global collection
Result	Longint	← Collection handle

Discussion

This command makes a deep (recursive) copy of the collection referenced by *inRef* and returns a handle to the copy. You can then use this handle with the other collection commands.

If the optional * parameter is given, the new collection is a global collection, otherwise the collection is local and will be cleared when the current script is finished executing.

When copying *inRef*, the following recursive algorithm is followed:

- 1 A shallow copy of the *inRef* is made.
- 2 The type of each item in the copy is checked.
- 3 If the item is a Longint and is a collection handle, the value is replaced with the result of a deep copy of that collection.

- 4 If the item is a Longint Array, each element of the array is checked to see if it is a collection, and if so the element is replaced with the result of a deep copy of that collection.

Warning: Circular collection references encountered by this command will cause Active4D (and 4D) to die a horrible death.

merge collections

version 3.0

merge collections(inRef1; inRef2 {...inRefN} {; *}) → Longint

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
*	*	→ Create a global collection
Result	Longint	← Collection handle

Discussion

This command merges all of the collections referenced by *inRef1...inRefN* and returns a handle to the merged collection. You can then use this handle with the other collection commands.

The merged collection is the union of all of the items in referenced collections. Because the collections are merged from left to right, if items two or more collections have the same key, the items in the collections to the right will overwrite those to the left.

If the optional * parameter is given, the new collection is a global collection, otherwise the collection is local and will be cleared when the current script is finished executing.

Example

```
// We want a merge of form variables and query params
c_longint($attributes)
$attributes := merge collections(form variables; query params)
```

clear collection

version 2`clear collection(inHandle)`

Parameter	Type	Description
<code>inHandle</code>	Longint	→ Collection handle

Discussion

This command deletes the collection and all of its items. After using this command, *inHandle* is no longer a valid collection handle.

Note: You may use this command on local collections, but there is usually not much point as local collections are automatically cleared when script execution ends.

deep clear collection

version 4.0`deep clear collection(inHandle)`

Parameter	Type	Description
<code>inHandle</code>	Longint	→ Collection handle

Discussion

This command deletes the collection and all of its items. After using this command, *inHandle* is no longer a valid collection handle.

Note: You may use this command on local collections, but there is usually not much point as local collections are automatically cleared when script execution ends.

When clearing *inHandle*, the following recursive algorithm is followed:

- 1 The type of each item is checked.
- 2 If the item is a Longint and is a collection handle, a deep clear of that collection is performed.
- 3 If the item is a Longint Array, each element of the array is checked to see if it is a collection, and if so a deep clear of that collection is performed.

Warning: Circular collection references encountered by this command will cause Active4D to die horribly.

get collection

version 2
modified version 4.0

```
get collection(inRef; inKey {; inIndex}) → <any>
```

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
inKey	Text	→ Key of collection item to retrieve
inIndex	Longint	→ Index of array element to retrieve
Result	<any>	← Value of collection item or ""

Discussion

This command searches the collection specified by *inRef* for the item with the key *inKey*.

If the item is found and *inIndex* is not specified, the item's value is returned. If the item value is an array, an empty string is returned.

If the item is found and an index is specified, the given array element is returned. If the index is out of range, an error is generated and execution is aborted.

If the item is not found, an empty string is returned.

Note: This command has been superseded by the simpler syntax:

```
collectionRef{inKey} or collectionRef{inKey}{inIndex}
```

get collection array

version 2
modified version 4.0

```
get collection array(inRef; inKey; outArray)
```

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
inKey	Text	→ Key of collection item to retrieve
outArray	Array	← Receives the array

Discussion

This command searches the collection specified by *inRef* for the item with the key *inKey*.

If *outArray* is not an array, an error is generated and execution is aborted.

If the item is found and its value is an array, *outArray* receives a copy. If *outArray* has not yet been defined, it is created with the same type as the source array.

If the item is found and its value is not an array, an error is generated and execution is aborted.

If the item is not found, *outArray* is resized to zero. If *outArray* has not yet been defined, it is created as a text array.

Note: This command has been superceded by the simpler syntax:

```
collectionRef{inKey}
```

If the resulting value is an array, it may be used with all of the array commands such as **append to array** and **DELETE ELEMENT**.

get collection array size

version 2
modified version 4.0

get collection array size(inRef; inKey) → Longint

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
inKey	Text	→ Key of collection item to check
Result	Longint	← Size of array

Discussion

This command searches the collection specified by *inRef* for the item with the key *inKey*.

If the item is found and its value is an array, the size of the array is returned.

If the item is found and its value is not an array, an error is generated and execution is aborted.

If the item is not found, zero is returned.

Note: This command has been superceded by the simpler syntax:

```
Size of array(collectionRef{inKey})
```

get collection item

version 2
modified version 4.0

get collection item(inRef; inKey) → Longint

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
inKey	Text	→ Key of collection item to retrieve
Result	Longint	← Iterator for collection

Discussion

This command searches the collection specified by *inRef* for the item with the key *inKey*.

If the item is found, an iterator reference for the item is returned.

If the item is not found, an empty iterator is returned. For information on empty iterators, see “Iterating Over a Collection” on page 102.

get collection item count

version 4.0

get collection item count(inRef; inKey) → Longint

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
inKey	Text	→ Key of form variable to check
Result	Longint	← Size of array

Discussion

This command searches the collection specified by *inRef* for the item with the key *inKey*.

If the item is found and contains an array, the size of the array is returned.

If the item is found and its value is not an array, 1 is returned.

If the item is not found, zero is returned.

get collection keys

version 2
modified version 4.0

get collection keys(inRef; outKeys)

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
outKeys	String/Text Array	← Receives the collection keys

Discussion

This command fills *outKeys* with all of the keys in the collection. If *outKeys* has not yet been defined, it is created as a string array.

If *outKeys* is defined but is not a string or text array, an error is generated and execution is aborted.

set collection

version 2

set collection(inHandle; inKey; inValue {; inKeyN; inValueN | inIndex})

Parameter	Type	Description
inHandle	Longint	→ Collection handle
inKey	Text	→ Key of collection item to store
inValue	<any>	→ Value to set for the given item
inIndex	Longint	→ Index of array element to set

Discussion

This command searches the collection specified by *inHandle* for the item with the key *inKey*.

If the item is found and its value is not an array, you may pass more than one key/value pair to set multiple items at once. If the item is found and its value is an array, you may pass an index to set an element of the array.

If the item is found and an index is not specified, the item's value is replaced with *inValue*.

If the item is found, its value is an array, and an index is specified, the given array element is set. If the index is out of range or the type of *inValue* is not assignment-compatible with the array, an error is generated and execution is aborted.

If the item is not found and an index is not specified, a new item is added to the collection with the given keys and values.

If the item is not found and an index is specified, an error is generated and execution is aborted.

Here is an example of **set collection** using multiple items:

```
$c := new collection
set collection($c; "name"; "Aparajita"; "age"; 40)
// Collection now contains two items
```

Note: This command has been superceded by the simpler syntax:

```
collectionRef{inKey} := inValue or collectionRef{inKey}{inIndex} := inValue
```

set collection array

version 2

```
set collection array(inHandle; inKey; inArray)
```

Parameter	Type	Description
inHandle	Longint	→ Collection handle
inKey	Text	→ Key of collection item to store
inArray	Array	→ Array to set for the given item

Discussion

This command searches the collection specified by *inHandle* for the item with the key *inKey*.

If *inArray* is not an array, an error is generated and execution is aborted.

If the item is found, its value is replaced with *inArray*.

If the item is not found, a new item is added to the collection with the given key and array.

Note: This command has been superceded by the simpler syntax:

```
collectionRef{inKey} := inArray
```

is a collection

version 2
modified version 4.0

is a collection(inRef) → Boolean

Parameter	Type	Description
inRef	<any>	→ Potential collection handle or iterator
Result	Boolean	← True if inRef is a collection handle

Discussion

This command tests a value to ensure it is a collection handle. Although Active4D tests the validity of collection handles in commands that take one, if the handle is not valid an error is generated and execution is aborted.

If you are unsure if a value is a collection handle and you don't want your script to be prematurely aborted, you should use this command to test the value before passing it to a collection command.

If the value is not a Longint, this command returns *False*.

collection has

version 2
modified version 4.0

collection has(inRef; inKey {; *}) → Boolean

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
inKey	Text	→ Key of collection item to test
*	*	→ Perform wildcard search
Result	Boolean	← True if key is in collection

Discussion

This command searches the collection specified by *inRef* for the item with the key *inKey*. If * is passed, *inKey* may contain 4D wildcard characters and they will be honored in the search.

If the item is found, *True* is returned, otherwise *False*.

If a collection value might be a non-text type, it is not sufficient to get its value and check for an empty string being returned. In such cases you should always use this command to test for the existence of a collection item, as shown here:

```
$c := new local collection

if ($c{"test"} # "")
  // This would break if "test" existed and was a number
end if

// The correct way
if (collection has($c; "test"))
  $test := $c{"test"}
else
  $test := "this is a test"
  $c{"test"} := $test
end if
```

count collection items

version 2
modified version 4.0

count collection items(inRef) → Longint

Parameter	Type	Description
inRef	Longint	→ Collection handle or iterator
Result	Longint	← Number of items in collection

Discussion

This command returns the number of items in the collection specified by *inRef*.

delete collection item

version 2

delete collection item(inHandle; inKey)

Parameter	Type	Description
inHandle	Longint	→ Collection handle
inKey	Text	→ Key of collection item to delete

Discussion

This command searches the collection specified by *inHandle* for the item with the key *inKey*. To delete more than one item, you may use a wildcard in the key. All items that match will be removed from the collection.

Cryptography

Active4D provides a few simple commands for doing basic encryption/decryption and hashing of text, in case you need to make aspects of your site more secure.

Each of the cryptography commands has a corresponding plugin command accessible from 4D, so that you can share encrypted data between 4D and Active4D.

The details of the various cryptographic algorithms employed are beyond the scope of this document. There is a wealth of information on cryptography online.

base64 decode

version 4.0
modified v5

base64 decode(inText {; inCharset {; * | inURLSafe}}) → Text | BLOB
base64 decode(inBlobData) → BLOB

Parameter	Type	Description
inText	Text	→ Base64 encoded text
inCharset	Text	→ Charset of decoded text
*	*	→ Use URL safe mode
inURLSafe	Boolean	→ True to use URL safe mode
Result	Text BLOB	← Original data
inBlobData	BLOB	→ Base64 encoded data
Result	BLOB	← Original data

Discussion

This command decodes text that was encoded with base 64 encoding (such as the **base64 encode** command).

If passed text and no charset, **base64 decode** returns a BLOB.

If passed text and a charset, **base64 decode** decodes the text, converts from the given charset to Unicode, and returns Unicode text.

If * is passed or *inURLSafe* is passed and is *True*, text is assumed to be in a URL safe form of base 64 encoding where '+' and '/' are replaced with '-' and '_' and the "=" padding at the end is stripped. If you do URL safe decoding, be sure the text was encoded in a URL safe form.

When text is passed, whitespace (carriage return, linefeed, space, tab) in the text is ignored. If the text contains invalid base64 characters, an empty string is returned.

If passed a BLOB, **base64 decode** returns a BLOB, because there is no guarantee that the decoded data contains valid Unicode.

base64 encode

version 4.0
modified v5

base64 encode(inText {; inCharset {; * | inURLSafe})) → Text
base64 encode(inBlobData) → Text

Parameter	Type	Description
inText	Text	→ Original text
inCharset	Text	→ Charset of encoded text
*	*	→ Use URL safe mode
inURLSafe	Boolean	→ True to use URL safe mode
Result	Text	← Encoded text
inBlobData	BLOB	→ Original data
Result	Text	← Encoded data

Discussion

This command encodes arbitrary bytes of binary data as a base 64 number which is represented by a letter or symbol in the 7-bit ASCII character set. The 7-bit ASCII nature of this text makes it easy to use in a URL, store in a database text field, etc., because it will never be changed due to character set conversions.

This command is primarily designed to be used in conjunction with **blowfish encrypt**, which returns 8-bit ASCII.

Because Unicode is a multibyte character set and base 64 is byte-oriented, Unicode text must be converted to a byte-oriented character set before it is encoded.

By default, Unicode text passed to **base64 encode** is converted to Mac Roman for backwards compatibility. You can specify another character set (such as UTF-8) by passing the *inCharset* parameter, which should be a valid IANA charset name. If the name is empty, it defaults to Mac Roman. If the name is not valid, an error is generated and execution is aborted.

Note: The charset you choose should be byte-oriented (e.g. UTF-8), not multibyte (e.g. UTF-16).

If * is passed or *inURLSafe* is passed and is *True*, a URL safe form of base 64 encoding is created where '+' and '/' are replaced with '-' and '_' and the "=" padding at the end is stripped. If you create URL safe encoding, be sure to do URL safe decoding as well.

base64 encode can take a BLOB as the first parameter, in which case the bytes of the blob are encoded directly and no charset is necessary.

Example

```
$id := string(session{"user.id"})
$enc := blowfish encrypt($id; "test"; "test"; "utf-8")
$qry := build query string(""); "u"; base64 encode($enc)
```

blowfish decrypt**version 4.0**
modified v5

blowfish decrypt(inBlob; inPassphrase {; inIV {; inCharset { *}}}) → Text | <none>

Parameter	Type	Description
inBlob	BLOB	→ Encrypted data to decrypt
inPassphrase	Text	→ Key used to encrypt data
inIV	Text	→ IV used to encrypt data
inCharset	Text	→ Charset of encrypted text
*	*	→ Decrypt in place
Result	Text <none>	← Encrypted text

Discussion

This command decrypts data that was encrypted with the Blowfish symmetric block cipher (such as the **blowfish encrypt** command). The passphrase and initialization vector (*inIV*) must be the same as those that were used to encrypt the text.

If * is not passed, the decrypted BLOB is converted to Unicode text. If *inCharset* is not passed in, the decrypted data is converted from Mac Roman for backwards compatibility. You can specify another character set (such as UTF-8) by passing *inCharset*, which should be a valid IANA charset name. If the name is empty, it defaults to Mac Roman. If the name is not valid, an error is generated and execution is aborted.

Note: The charset you choose should be byte-oriented (e.g. UTF-8), not multibyte (e.g. UTF-16).

If * is passed, the data is decrypted in place and nothing is returned.

blowfish encrypt

version 4.0
modified v5

blowfish encrypt(inText; inPassphrase {; inIV {; inCharset}}) → BLOB
blowfish encrypt(inBLOB; inPassphrase {; inIV {; inCharset {; *}}}) → BLOB | <none>

Parameter	Type	Description
inText	Text	→ Text to encrypt
inPassphrase	Text	→ Key to encrypt/decrypt text
inIV	Text	→ Seed text for encryption
inCharset	Text	→ Character set to convert to
Result	BLOB	← Encrypted data
inBLOB	BLOB	→ Data to encrypt
inPassphrase	Text	→ Key to encrypt/decrypt data
inIV	Text	→ Seed text for encryption
inCharset	Text	→ Character set to convert to
*	*	→ Encrypt in place
Result	BLOB	← Encrypted data

Discussion

This command encrypts *inText* or *inBLOB* with the Blowfish symmetric block cipher.

Blowfish encryption and decryption in Active4D use a variable length key between 1 and 56 bytes and an eight-byte IV, or *initialization vector*. After conversion to the target character set, both the key and IV are truncated to the maximum length or padded with spaces to the minimum length.

Text to be encrypted is converted from Unicode to the target character set, then padded with PKCS#5 padding to an even multiple of eight bytes before encryption. If you are encrypting and decrypting within Active4D, you don't need to know this, but if you are sharing encrypted data with other software, you need to know the encryption format.

inPassphrase is the key with which you encrypt and decrypt the text. It must be non-empty.

In addition, you may supply an initialization vector, or seed text, in the *inIV* parameter. The initialization vector (IV) is used as the seed text which the encryption algorithm uses. You may leave it empty if you wish.

By default, the text, passphrase and IV are converted to Mac Roman for backwards compatibility. You can specify another character set (such as UTF-8) by passing the *inCharset* parameter, which should be a valid IANA charset name. If the name is empty, it defaults to Mac Roman. If the name is not valid, an error is generated and execution is aborted.

Note: The charset you choose should be byte-oriented (e.g. UTF-8), not multibyte (e.g. UTF-16).

Typically the IV is used to add a bit of randomness to the encrypted text, to make it harder for potential crackers to determine the passphrase. Because the IV must be known in order to decrypt, it is usually added to the encrypted text and then retrieved when the text is decrypted.

For example, you could do something like this:

```
// encryption
$enc := blowfish encrypt(string(milliseconds); "ivpass"; "")
$iv := substring(base64 encode($enc); 1; 8)
$enc := blowfish encrypt("Some text"; "secret pass"; $iv; "utf-8")
$encText := $iv + base64 encode($enc)
$queryValue := url encode query($encText)

// decryption
$encText := _query{"q"} // already url decoded by Active4D
$iv := substring($encText; 1; 8) // separate IV
$encText := substring($encText; 9)
$encText := base64 decode($encText; "utf-8")
$text := blowfish decrypt($encText; "secret pass"; $iv)
```

Like I said, you *could* do something like the above if you are paranoid or are operating in an environment where your data is so valuable that it is worth it for someone who has both significant knowledge of cryptographic algorithms and significant computing resources to attempt to crack your encryption.

In other words, for the vast majority of us, it is quite sufficient to use either no IV or the same IV all of the time.

If you pass a BLOB as the first parameter and do not pass *, an encrypted BLOB is returned. If you pass a BLOB and *, the BLOB is encrypted in place and nothing is returned.

Note: **blowfish encrypt** always encrypts to a BLOB in Active4D v6 because encryption actually creates a bunch of numbers which cannot reliably be converted to valid Unicode text.

md5 sum**version 4.0
modified v5**

md5 sum(inData {; inCharset}) → Text
 md5 sum(inData) → Text

Parameter	Type		Description
inText	Text	→	Original data
inCharset	Text	→	Charset to convert to
Result	Text	←	MD5 hash
inData	BLOB	→	Original data
Result	Text	←	MD5 hash

Discussion

This command performs an MD5 hash of *inData*, which results in a 128-bit (16-byte) checksum that is returned as a 32-byte hexadecimal number.

MD5 is a one-way algorithm designed to uniquely identify a sequence of bits in a secure manner. You cannot reconstruct the original data from the resulting MD5 checksum.

If *inData* is text and *inCharset* is not passed, the text is converted to Mac Roman for compatibility with Active4D v4.x. If *inCharset* is passed and is empty, it defaults to Mac Roman. Otherwise it should be a valid IANA character set name or alias. If the name is invalid, an error is generated and execution is aborted.

For a complete list of valid character set names, see:

<http://demo.icu-project.org/icu-bin/convexp?s=IANA&s=ALL>

You may also pass a BLOB to this command to calculate a hash on arbitrary data.

Database

Active4D adds a few database commands which make it easier to dynamically get pointers to tables and fields.

Trigger Errors

If your database uses triggers which return your own error numbers (-15000 to -32000), those errors are available from Active4D for the following triggers:

- On saving new record
- On saving an existing record
- On deleting a record

If a trigger error is returned from a command that invokes one of those triggers (**SAVE RECORD**, **DELETE RECORD** and **DELETE SELECTION**), you can access the error number in the variable *A4D_Error*. You may also check the value of the *OK* variable to see if the command executed successfully.

For example, the following code save a record and then checks for a trigger error, warning the user if something went wrong.

```
save record([people])

if (A4D_Error # 0)
    writebr("An error occurred saving the record: " + A4D_Error)
end if
```

Field name

**version 1
modified v4.0**

Field name(inFieldPtr {; *} | inTableNum; inFieldNum {; *}) → Text

Parameter	Type	Description
inFieldPtr	Pointer	→ Field pointer
inTableNum	Number	→ Number of field's table
inFieldNum	Number	→ Field's number
*	*	→ Return full [table]field ref
Result	Text	← Field name

Discussion

Active4D enhances the 4D version of the **Field name** command with an extra option. If * is passed as the last parameter, the full *[table]field* reference will be returned.

get field numbers

version 1

get field numbers(inReference; outTableNum; outFieldNum)

Parameter	Type	Description
inReference	Text	→ Table or field reference as text
outTableNum	Longint	← Receives the table number
outFieldNum	Longint	← Receives the field number

Discussion

This command resolves a table or field reference in the form "[table]" or "[table]field" into the corresponding table and field numbers. It allows you to dynamically build a table or field reference as a string and then resolve it.

If *inReference* is not a valid table or field reference, both *outTableNum* and *outFieldNum* will be zero.

If the reference is valid but only a table is passed, *outFieldNum* will be zero.

get field pointer

version 1

get field pointer(inTable; inField | inReference) → Pointer

Parameter	Type	Description
inTable	Number Pointer	→ Table number of pointer
inField	Text	→ Field name (no table)
inReference	Text	→ Table or field reference as text
Result	Pointer	← Pointer to table or field

Discussion

The first form of this command takes two parameters. The first parameter is a table number or pointer, and the second is a field name (without a table name). If *inField* is a valid field name within the table referenced by *inTable*, a pointer to the field is returned, otherwise a nil pointer is returned.

The second form of this command takes one parameter, which is a table or field reference in the form "[table]" or "[table]field". If *inReference* is a valid table or field reference, a pointer to that table or field is returned. If *inReference* is not valid, a nil pointer is returned.

QUERY

version 1
modified v2, v6.1r2

QUERY(inTable; {; inConjunction}; inField; inComparator; inValue {; *})

Parameter	Type	Description
inTable	[table]	→ Main table on which to query
inConjunction	Logical Text	→ Conjunction with previous line
inField	[table]field	→ Field on which to query
inComparator	Operator Text	→ How to compare field with value
inValue	<any>	→ Value to compare against field

Discussion

Active4D supports 4D's extended **QUERY** syntax with its own extension:

- *inConjunction* may be used on the first line of a built query, but it is ignored.
- *inConjunction* may be any unary expression that resolves to "&", "|" or "#". This is an extension that 4D does not support.
- *inConjunction* may be omitted in multi-line queries, and it will default to &.
- *inComparator* and *inValue* may be separate arguments.

- *inComparator* may be any expression that resolves to a valid value comparator ("=", "#", "<", "<=", ">", ">=" or "%").

QUERY BY FORMULA

version 1
modified v5

QUERY BY FORMULA({*; } inTable; <expression>)

Parameter	Type	Description
*	*	→ Execute within 4D's context
inTable	[table]	→ Main table on which to query
expression	Boolean	→ Boolean expression

Discussion

If * is not passed, this command differs from the 4D version of **QUERY BY FORMULA** in that *expression* is evaluated by Active4D and must be valid Active4D code. This allows you to call Active4D methods and use local variables in the expression, as well as reference collections like *_form*, *_query*, etc.

If * is passed, this command is executed on the server within 4D's context, which means it may execute faster, but you will have no access to Active4D's variables, collections and methods.

QUERY SELECTION

version 1
modified v2, v6.1r2

QUERY SELECTION(inTable; {; inConjunction}; inField; inComparator; inValue {; *})

Parameter	Type	Description
inTable	[table]	→ Main table on which to query
inConjunction	Logical Text	→ Conjunction with previous line
inField	[table]field	→ Field on which to query
inComparator	Operator Text	→ How to compare field with value
inValue	<any>	→ Value to compare against field

Discussion

See the discussion of **QUERY**.

QUERY SELECTION BY FORMULA

version 4.0
modified v5

QUERY SELECTION BY FORMULA({*; } inTable; <expression>)

Parameter	Type	Description
*	*	→ Execute within 4D's context
inTable	[table]	→ Main table on which to query
expression	Boolean	→ Boolean expression

Discussion

See the discussion of **QUERY BY FORMULA**.

SET QUERY DESTINATION

version 1

SET QUERY DESTINATION(inDestination {; inValue})

Discussion

This command differs from the 4D version if *inDestination* is *Into variable*:

- Local variables are not allowed.
- The process or interprocess variable specified must already be defined, but you can define them yourself with a compiler declaration within Active4D.

Table name

version 1
modified v4.0

Table name(inTableNum {; *} | inTablePtr {; *}) → Text

Parameter	Type	Description
inTableNum	Number	→ Table number
inTablePtr	Pointer	→ Pointer to table
*	*	→ Return [tablename]
Result	Text	← Field name

Discussion

Active4D enhances the 4D version of the **Table name** command with an extra option. If * is passed as the last parameter, the table name will be returned enclosed by square brackets ([]).

Date and Time

UTC Commands

In an internet-based application, your users may be anywhere in the world. It is important to be able to keep track of various times in a consistent way.

Coordinated Universal Time, otherwise known as UTC or GMT, is an international standard for coordinating times from different time zones. By storing your dates and times in UTC, you can be sure that date and time comparisons are accurate.

Active4D provides a suite of commands that allow you to easily convert between local time and UTC. These conversions are based on the system settings for the current time zone and whether or not your location within the time zone uses Daylight Savings Time.

Date

v5 (modified 4D)

Date(inYear; inMonth; inDay) → Date

Parameter	Type		Description
inYear	Number	→	Year of date
inMonth	Number	→	Month of date
inDay	Number	→	Day of date
Result	Date	←	Full date

Discussion

The 4D **Date** command has been extended to allow you to construct a date from a year, month and day. This allows you to quickly create a date without having to play tricks like constructing a string and then parsing it, or using the idiom:

```
// old way, obscures your intention
$date := add to date(!00/00/00!; $year; $month; $day)

// new way, much clearer
$date := date($year; $month; $day)
```

day of year

version 2

day of year(inDate) → Number

Parameter	Type		Description
inDate	Date	→	A date
Result	Number	←	Day number within the date's year

Discussion

This command returns the serial day number within the year represented by *inDate*. For example:

```
$day := day of year(!01/01/01!) // $day = 1
$day := day of year(!12/31/01!) // $day = 365
```

get utc delta

version 2
modified v5

```
get utc delta(outHours; outMinutes)
get utc delta → Number
```

Parameter	Type	Description
outHours	Number variable	← Hour difference between local/UTC time
outMinutes	Number variable	← Minute difference between local/UTC time
Result	Number	← Minute difference between local/UTC time

Discussion

In the first form of this command, it returns the number of hours and minutes that would have to be *subtracted* from the local time to get a UTC time. Note that local times west of Greenwich Mean Time will result in negative *outHours*. Subtracting this negative number results in adding that many hours to the local time.

In the second form of this command, you pass no parameters and it returns the number of minutes that would have to be subtracted from the local time to get a UTC time. In other words, it is the equivalent of:

```
get utc delta($hours; $minutes)
$minutes += $hours * 60
```

local datetime to utc

version 2

```
local datetime to utc(ioDate; ioTime)
```

Parameter	Type	Description
ioDate	Date variable	↔ Date to convert
ioTime	Time variable	↔ Time to convert

Discussion

This command converts *ioDate* and *ioTime* from a date and time in the local time zone to UTC.

local time to utc

version 2local time to utc(*inTime*) → Time

Parameter	Type	Description
<i>inTime</i>	Time	→ A time
Result	Time	← Converted time

Discussion

This command converts *inTime* from a time in the local time zone to UTC. The UTC time is normalized to fall in the range 00:00:00 to 23:59:59.

The UTC date may be before or after the local date. If you need the date as well, use the **local datetime to utc** command.

utc to local datetime

version 2utc to local datetime(*ioDate*; *ioTime*)

Parameter	Type	Description
<i>ioDate</i>	Date variable	↔ Date to convert
<i>ioTime</i>	Time variable	↔ Time to convert

Discussion

This command converts *ioDate* and *ioTime* from a date and time in UTC to the local time zone.

utc to local time

version 2utc to local time(*inTime*) → Time

Parameter	Type	Description
<i>inTime</i>	Time	→ A time
Result	Time	← Converted time

Discussion

This command converts *inTime* from a time in UTC to a time in the local time zone. The local time is normalized to fall in the range 00:00:00 to 23:59:59.

The local date may be before or after the UTC date. If you need the date as well, use the **utc to local datetime** command.

week of year**version 2**

week of year(inDate) → Number

Parameter	Type	Description
inDate	Date	→ A date
Result	Number	← Week number within the date's year

Discussion

This command returns the serial week number within the year represented by *inDate*. For example:

```
$week := week of year(!01/01/01!) // $week = 1
$week := week of year(!12/31/01!) // $week = 53
```

Note: The week number for dates in the first week of the year may actually be 53. For more information, see the “WEEK_OF_YEAR” note at:

<http://userguide.icu-project.org/datetime/calendar#TOC-Disambiguation>

Debugging

No development environment would be complete without debugging tools. Although Active4D does not yet have an interactive debugger, it provides a number of debugging tools that allow you quickly and easily view what is going on “inside” your scripts.

Most of the debugging tools are actually part of the standard libraries and 4D shell code distributed with Active4D. The commands in this section are for the most part low level commands used by the debugging tools.

For more information on Active4D’s debugging tools, see Chapter 13, “Debugging.”

current library name

version 4.0

current library name → Text

Parameter	Type	Description
Result	Text	← Library name

Discussion

This command returns the name of the library in which execution is taking place at the time this command is called. If this command is executed outside of a library method, an empty string is returned.

current line number

version 2

current line number → Number

Parameter	Type	Description
Result	Number	← Line number

Discussion

This command returns the line number of the currently executing method. If the command is not executed within a method, the line number is the line number of the source file in which the command appears. If the command is executed within a method, the line number returned is the line number within the body of the method, not including the method declaration.

current method name

version 2

current method name{*} → Text

Parameter	Type	Description
*	*	→ Show full method descriptor
Result	Text	← Method descriptor

Discussion

This command has two forms. If the * is not passed, the name of the current method is returned. If you are currently executing outside of an Active4D method, an empty string is returned.

If the * is passed in, a full descriptor of the current execution context is returned as follows:

Context	Descriptor
File	Full URL-style path to the file
Inline method	Method "<method>"
Library method	Method "<lib.method>"<argument list>

get call chain

version 3

get call chain(outChain)

Parameter	Type	Description
outChain	Text Array	← Receives the call chain

Discussion

This command returns the names of all of the methods in the call chain from the point at which it is executed.

outChain must be a local variable, but it need not be defined before calling this command.

The chain is represented with the currently executing method in element 1, followed by the caller in element 2, and so on. The names are in the form *library.method*, except for the top level execution scope, which is called "main". For inline methods (not declared in a library), the library name will be "global".

The values in the *outChain* array are suitable for passing to the **local variables** command.

library list

version 4.0

library list(outList)

Parameter	Type	Description
outList	Text Array	← Receives the list of imported libraries

Discussion

This command returns the names of all of the libraries that have been imported. If *outList* does not exist or is not a Text array, it will created as or converted into a Text array.

write to console

version 2

```
write to console(inValue {; inFormat})
```

Parameter	Type	Description
inValue	<any>	→ Value to write to the response buffer
inFormat	Text Number	→ Format to use when converting to text

Discussion

This command is functionally equivalent to the write command, but instead of appending its output to the response buffer, it appends its output to the Active4D debugging console.

For more information on the Active4D debugging console, see “Using the Debugging Console” on page 606.

Error Handling

Bad things happen to good people. When things do go wrong, Active4D is there to help you find out why.

The commands in this section control the various aspects of error handling in Active4D, which is discussed in Chapter 14, "Error Handling." It is recommended that you read that chapter before reading the command descriptions in this section.

get error page

version 1

get error page → Text

Parameter	Type	Description
Result	Text	← Root-relative path to error page

Discussion

This command returns the root-relative error page path for the currently executing script. The default error page is specified by the “error page” option in Active4D.ini.

get error status

version 1

get error status → Number

Parameter	Type	Description
Result	Number	← The status that triggered an error

Discussion

Within the context of an error handler, this command returns the response status that triggered the handler. For example, if an execution error occurs and there is a custom error handler, within the handler the response status will be 200 (OK), but **get error status** will return 500 (Internal Server Error).

If executed outside of an error handler, this command will always return zero.

get http error page

v6.0

get http error page → Text

Parameter	Type	Description
Result	Text	← Root-relative path to HTTP error page

Discussion

This command returns the root-relative http error page path. The default http error page is specified by the “http error page” option in Active4D.ini.

get log level

version 1

get log level → Number

Parameter	Type	Description
Result	Number	← The current log level

Discussion

This command returns the current log level, which determines which errors or messages are logged.

in error

version 1

in error → Boolean

Parameter	Type	Description
Result	Boolean	← True if executing an error page

Discussion

This command returns *True* if an error page is currently being executed, *False* if not.

log message

version 1
modified v6.4r1

```
log message(inMessage {; inIsError})
```

Parameter	Type	Description
inMessage	<Any>	→ Message to append to the log file
inIsError	Boolean	→ True to mark as an error message

Discussion

This command converts *inMessage* to a string and then sends it to the Active4D log file if the *A4D Log User Messages* flag is set in the current log level. Whereas log messages generated by Active4D are marked with "Active4D:" after the date and time, log messages generated by this command are marked by "DB:". This allows you to easily filter the log entries by their source.

Because this command has no effect if the *A4D Log User Messages* flag is not set, you can set the flag during development and use this command to write debugging messages, then turn off the flag during production to eliminate the debug messages.

If *inIsError* is passed and is *True*, the message will be flagged in the log as an error.

set error page

version 1

```
set error page(inPath)
```

Parameter	Type	Description
inPath	Text	→ Root-relative path to error page

Discussion

After executing this command, any Active4D errors that occur will cause the given script to be executed. To use the default error page as specified by the "error page" option in *Active4D.ini*, pass an empty string.

The path given should be a URL-style root-relative path. A leading '/' is ignored. In addition, the path must contain no native directory separators and no directory movement (*/../*).

The path is not validated when this command is executed, but rather when Active4D attempts to load the given error page. Thus it is up to you to ensure that *inPath* is valid.

set http error page

v6.0

```
set http error page(inPath)
```

Parameter	Type	Description
inPath	Text	→ Root-relative path to HTTP error page

Discussion

After executing this command, any HTTP errors that occur will cause the given script to be executed. To use the default HTTP error page as specified by the “http error page” option in Active4D.ini, pass an empty string.

The path given should be a URL-style root-relative path. A leading ‘/’ is ignored. In addition, the path must contain no native directory separators and no directory movement (/../).

The path is not validated when this command is executed, but rather when Active4D attempts to load the given error page. Thus it is up to you to ensure that *inPath* is valid.

Note: This command operates on a global basis and will affect all subsequent requests.

set log level

version 1

```
set log level(inLevel)
```

Parameter	Type	Description
inLevel	Number	→ Set of bit flags

Discussion

This command sets the current log level, which determines which errors or messages are logged.

File Uploads

The HTTP specification allows for binary objects to be included as part of a posted form, and the HTML specification implements this by providing an input field of type "file". When such a field is placed on a form, the browser shows a "Browse" button, enabling the user to select a file for uploading.

When the form is posted, the form's fields and the file's contents are specially encoded and sent to the server. Active4D transparently handles the decoding of this type of request for you.

How File Upload Works

When a user posts a form with an uploaded file, before code execution starts Active4D saves the file to a directory called "Active4D uploads" in the default directory (see "The Default Directory" on page 42). Note that for security reasons, you may not reference the file directly in that directory.

If the size of the uploaded file would exceed the maximum request size specified by the "max request size" option in Active4D.ini, the uploaded file is not saved and Active4D returns the status *413 Request Entity Too Large*.

The uploaded file is given the name `<sequence><extension>`, where `<sequence>` is a number starting from 1 which is incremented each time a file is uploaded, and `<extension>` is the file extension (if any) of the upload filename as it was on the client's machine. If a file already exists in the "Active4D uploads" directory with the proposed name, Active4D increments the sequence number and tries again until the name is unique.

Referencing File Uploads

Within your code, you reference uploaded files by the name of the file field on the form that was posted. For example, if your HTML form contains this field definition:

```
<input type="file" name="upload" />
```

you would reference the uploaded file like this:

```
$size := get upload size("upload")
```

Every file upload command takes a reference as the first parameter.

Given an upload reference, you can find out information about the file, copy it from the upload directory to another directory, or save it to the database. If you pass a reference to a non-existent upload to an upload command, *OK* is set to zero and *A4D_Error* is set to *-43 (File Not Found)*.

The Importance of Filename Extensions

It is important that you know the type of a file. For example, if you are requesting a graphic, you want to ensure that a graphic file is uploaded, not a text file.

Most browsers rely on filename extensions to determine the type of a file before uploading it. The presence and accuracy of this extension is critical. Thus you should encourage your users to upload files only with proper extensions.

If you want you can check the extension of the upload and reject the upload if it is incorrect or missing.

Upload Auto-Deletion

If an error occurs during script execution, you may lose track of uploads from the current request. To prevent the accumulation of orphaned uploads, uploads are automatically deleted from the “Active4D uploads” directory when the current script finishes executing, even if there was an error.

If want to retain a copy of the upload, before the script finishes executing, you must use on of the following:

- **copy upload** to copy the upload out of the “Active4D uploads” directory
- **upload to blob** to copy the upload to a blob
- **save upload to field** to save the upload to a database field

copy upload

version 2

copy upload(inReference; inDestPath)

Parameter	Type	Description
inReference	Text	→ Name of file field on form
inDestPath	Text	→ Directory and filename of destination

Discussion

This command copies the referenced upload to the directory and filename given by the native or URL-style path *inDestPath*. If *inDestPath* is absolute, it is considered a full path. If *inDestPath* is relative, it is relative to the default directory.

If a file already exists with the same path, it is overwritten. If the copy succeeds, *OK* is set to 1. Otherwise *OK* is set to zero and *A4D_Error* contains the error code.

As with all document commands, *inDestPath* must be within either the web root directory or a directory in the “safe doc dirs” path list in Active4D.ini. For information on document paths, see “Document Paths” on page 387. For more information on the “safe doc dirs” path list, see “Misusing Document Commands” on page 51.

count uploads

version 2

count uploads → Number

Parameter	Type	Description
Result	Number	← Number of files uploaded

Discussion

This command returns how many files were actually uploaded in the current request. Even though a file field may be on a form, the user may post the form without uploading a file.

get upload content type

version 2

get upload content type(inReference) → Text

Parameter	Type	Description
inReference	Text	→ Name of file field on form
Result	Text	← MIME type of file

Discussion

This command returns the MIME type of the given upload, which is determined by its filename extension. For example, if the uploaded file was called "logo.gif", this command will return "image/gif".

If no such upload exists, an empty string is returned.

get upload encoding

version 2

get upload encoding(inReference) → Text

Parameter	Type	Description
inReference	Text	→ Name of file field on form
Result	Text	← Transfer encoding of file

Discussion

This command returns the transfer encoding of the given upload, if it was provided by the client browser (which is usually not the case).

If a transfer encoding was not provided or no such upload exists, an empty string is returned.

get upload extension

version 2

get upload extension(inReference) → Text

Parameter	Type	Description
inReference	Text	→ Name of file field on form
Result	Text	← Filename extension of file

Discussion

This command returns the filename extension (including the dot) of the given upload.

If the upload has no extension or there is no such upload, an empty string is returned.

get upload remote filename

version 2

get upload remote filename(inReference) → Text

Parameter	Type	Description
inReference	Text	→ Name of file field on form
Result	Text	← Filename of file

Discussion

This command returns the filename of the given upload as it exists on the client's machine.

If no such upload exists, an empty string is returned.

Note: Some browsers return the full path to the remote file, some just return the filename. You can use the **filename of** command to be sure you have only the filename.

get upload size

version 2

get upload size(inReference) → Number

Parameter	Type	Description
inReference	Text	→ Name of file field on form
Result	Number	← Size in bytes of file

Discussion

This command returns the size in bytes of the given upload.

If no such upload exists, -1 is returned. You can use this to test whether the given file was actually uploaded.

save upload to field

version 2

save upload to field(inReference; inFieldRef | inFieldPtr)

Parameter	Type	Description
inReference	Text	→ Name of file field on form
inFieldRef	[table]field	→ Full field reference of destination
inFieldPtr	Field pointer	→ Pointer to destination field

Discussion

This command saves the given upload to the field identified either by a field reference or by a field pointer.

If the given upload does not exist, the destination field is not a BLOB field, or if the save fails in any way, *OK* is set to zero. If the save succeeds, *OK* is set to 1.

Note: Beginning with v6.0r10, you can assign the result of **upload to blob** directly to the field, although that will result in an extra copy of the file's contents in memory.

upload to blob

v6.0r10

upload to blob(inReference)

Parameter	Type	Description
inReference	Text	→ Name of file field on form
Result	BLOB	← Contents of file in a BLOB

Discussion

This command returns the contents of the given upload as a BLOB.

If the given upload does not exist or memory for the BLOB cannot be allocated, *OK* is set to zero and an empty BLOB is returned. Otherwise, *OK* is set to 1.

Note: Beginning with v6.0r10, you can assign the result this method directly to a field instead of using **save upload to field**, although doing so will result in an extra copy of the file's contents in memory.

Form Variables

When a form is posted, Active4D puts the form variable names and their associated values into a collection. You can access this collection in your scripts. The form variables collection is read-only, and follows the same pattern as all read-only collections.

When Form Variables Are Query Params (and vice versa)

If the configuration option “parameter mode” is set to “query params” in Active4D.ini, form variables will not go into the form variables collection, but into the query params collection. Likewise, if the “parameter mode” option is set to “form variables”, query parameters will go into the form variables collection.

Posting JSON Data

Normally, item values in the form variables collection are either text or text arrays. However, if a form is posted with the Content-Type “application/json”, and the “parse json request” configuration option is true, and the “parameter mode” configuration option is *not* set to “query params”, Active4D parses the form data (in the same way as the **parse json** command) and places the result in the form variables collection with the key “_json_”. If the “parse json request” configuration option is false, the JSON data is considered raw data.

Posting Raw Data

Normally, item values in the form variables collection are either text or text arrays. However, if a form is posted which contains raw data (such as XML) and no form variables, and the “parameter mode” configuration option is *not* set to “query params”, Active4D creates a single BLOB item in the form variables collection with the key “_data_”.

Multiple-choice Form Fields

When a multiple-choice list is placed on a form and the user selects more than one item in the list, one form variable for each selected item is sent as part of the posted form. In such cases, Active4D creates a text array to hold the selected items, the key of which is the choice list’s form name.

_form**version 3.0**`_form` → Longint

Parameter	Type	Description
Result	Longint	← Iterator reference

Discussion

This command is an alias for the **form variables** command.

form variables**version 2**`form variables` → Longint

Parameter	Type	Description
Result	Longint	← Iterator reference

Discussion

This command returns an iterator to the first item in the form variables collection.

For more information on iterators, see “Iterators” on page 214.

form variables has**version 2**`form variables has(inKey {; *})` → Boolean

Parameter	Type	Description
<code>inKey</code>	Text	→ Key of item to test
*	*	→ Perform wildcard search
Result	Boolean	← True if key is in collection

Discussion

This command searches the form variables collection for the item with the key `inKey`. If * is passed, `inKey` may contain 4D wildcard characters and they will be honored in the search.

If the item is found, *True* is returned, otherwise *False*.

get form variable

version 2

get form variable(inKey {; inIndex}) → Text | BLOB

Parameter	Type	Description
inKey	Text	→ Key of form variable to retrieve
inIndex	Longint	→ Index of multiple-choice list value to retrieve
Result	Text BLOB	← Value of form variable or ""

Discussion

This command searches the form variable collection for the item with the key *inKey*.

If the item is found and an index is not specified, the item's value is returned. If the item value is an array of multiple choices, the first multiple-choice value is returned. Note that this behavior is different than using the syntax:

```
$value := _form{"multiple_choice_field"} // Returns longint
```

When using the collection indexing syntax, a reference to an array with no index returns the array itself, which when assigned results in the current element of the array, which is a longint.

If the item is found and an index is specified, the given multiple choice value is returned. If the index is out of range, an error is generated and execution is aborted.

If the item is not found, an empty string is returned.

Note: For scalar values this command has been superseded by the simpler syntax:

```
_form{inKey} or _form{inKey}{inIndex}
```

get form variable choices

version 2

get form variable choices(inKey; outArray)

Parameter	Type	Description
inKey	Text	→ Key of form variable to retrieve
outArray	String/Text Array	← Receives the array of choices

Discussion

This command searches the form variables collection for the item with the key *inKey*.

If *outArray* is defined but is not a string or text array, an error is generated and execution is aborted. If *outArray* was not defined, it is created as a text array.

If the item is found and has multiple-choice values in an array, *outArray* is set to a copy of the multiple-choice values.

If the item is found and its value is not an array of multiple-choice values, *outArray* is set to a single element containing the form variable value.

If the item is not found, *outArray* is resized to zero elements.

Note: This command is not really necessary, as you can directly refer to a multiple-choice array with the syntax:

```
_form{inKey}
```

For more information see “Referencing Collection Values” on page 101.

get form variable count

version 2

get form variable count(inKey) → Longint

Parameter	Type	Description
inKey	Text	→ Key of form variable to check
Result	Longint	← Size of array

Discussion

You use this command to check how many values were selected in a multiple-choice list on a form.

This command searches the form variables collection for the item with the key *inKey*.

If the item is found and has multiple-choice values in an array, the size of the array is returned.

If the item is found and its value is not an array of multiple-choice values, 1 is returned.

If the item is not found, zero is returned.

get form variables

version 2

```
get form variables(outKeys {; outValues})
get form variables{*; inKeyFilter} → Text
```

Parameter	Type	Description
outKeys	String/Text Array	← Receives the collection keys
outValues	String/Text Array	← Receives the collection values
*		→ Indicates a filter is being used
inKeyFilter	Text	→ Keys are matched against this
Result	Text	← Concatenation of keys and values

Discussion

This command has two forms. The first form fills *outKeys* and *outValues* with all of the keys and values in the form variables collection. If a value is a multiple-choice array, the first element of the array is put into the *outValues* array.

If *outKeys* was not defined, it is created as a string array. If *outValues* was not defined, it is created as a text array.

If *outKeys* is defined but is not a string or text array, an error is generated and execution is aborted.

The second form of the command returns a concatenation of the form variables in the form "key1=value1;key2=value2", suitable for use as a query string. If a form variable is a multiple-choice array, all of the array values are included in the concatenation.

The keys and values are converted to UTF-8 and URL encoded.

You may optionally pass in a string which will be matched against items whose key matches the string. Wildcards are allowed in the filter string. To *include* items that match the filter, prefix it with '+'. To *exclude* items that match the filter, prefix it with '-'. If there is no prefix, it is assumed to be an inclusion filter.

count form variables

version 2

```
count form variables → Longint
```

Parameter	Type	Description
Result	Longint	← Number of items in collection

Discussion

This command returns the number of items in the form variables collection.

Globals

In the course of programming a web site, you may need to store application-wide information. Because each HTTP request is handled in a separate process, this means you must have storage that is accessible across processes.

You could define interprocess variables in 4D and access them in Active4D. However, this means you must modify the database if you need more interprocess variables in your web site. The whole premise of Active4D is to insulate the database from changes to your web site, so this is not the optimal solution.

Active4D provides a special global collection which can be accessed in any Active4D script. By using this collection, you no longer need to use 4D interprocess variables.

Locking and Unlocking the Globals

Because the global collection is accessible to any executing process, you must ensure that only one process at a time can change it, just as you would with an interprocess array in 4D.

In 4D you would accomplish this with a global semaphore. Active4D provides a custom lock specifically for the global collection. If you are setting more than one global collection item, you should surround the code with the commands **lock globals** and **unlock globals**.

Here is an example that would appear in the *On Application Start* and *On Session Start* methods in the Active4D library:

```
method "On Application Start"
  // We don't need to lock globals here
  // because it happens at startup
  set global("last visit"; ?00:00:00?)
  set global("visitors"; 0)
end method

method "On Session Start"
  lock globals // Make sure we have exclusive access
  set global("last visit"; current time)
  set global("visitors"; get global("visitors") + 1)
  unlock globals // Let other processes have access
  set session("visit start"; current time)
end method

method "On Session End"
  set global("visitors"; get global("visitors") - 1)
end method
```

Note that in the *On Session End* method, the globals are not being locked. Active4D automatically locks the globals within the **set global** command, so for a single **set global** call it is not necessary to lock the globals.

In general, you should lock the globals only when necessary and for as few lines of code as possible. While they are locked, any other processes that try to access the globals will be suspended. If this happens often enough it can have an adverse effect on the performance of your web site.

Note: To prevent a potential deadlock condition, Active4D releases the globals lock at the end of script execution and at the end of the following special methods:

On Application Start, On Application End, On Session Start, On Session End

globals

version 2

globals → Longint

Parameter	Type	Description
Result	Longint	← Iterator reference

Discussion

This command returns an iterator to the first item in the global collection.

For more information on iterators, see “Iterators” on page 214.

globals has

version 2

globals has(inKey) → Boolean

Parameter	Type	Description
inKey	Text	→ Key of item to test
Result	Boolean	← True if key is in collection

Discussion

This command searches the globals collection for the item with the key *inKey*. If the item is found, *True* is returned, otherwise *False*.

get global

version 2

get global(inKey {; inIndex}) → <any>

Parameter	Type	Description
inKey	Text	→ Key of global value to retrieve
inIndex	Longint	→ Index of array element to retrieve
Result	<any>	← Value of global or ""

Discussion

This command searches the globals collection for the item with the key *inKey*.

If the item is found and *inIndex* is not specified, the item’s value is returned. If the item value is an array, an empty string is returned.

If the item is found and an index is specified, the given array element is returned. If the index is out of range, an error is generated and execution is aborted.

If the item is not found, an empty string is returned.

Note: For scalar values this command has been superseded by the simpler syntax:

```
globals{inKey} or globals{inKey}{inIndex}
```

get global array

version 2

```
get global array(inKey; outArray)
```

Parameter	Type	Description
inKey	Text	→ Key of global item to retrieve
outArray	Array	← Receives the array

Discussion

This command searches the global collection for the item with the key *inKey*.

If *outArray* is not an array, an error is generated and execution is aborted.

If the item is found and its value is an array, *outArray* receives a copy. If *outArray* has not yet been defined, it is created with the same type as the source array.

If the item is found and its value is not an array, an error is generated and execution is aborted.

If the item is not found, *outArray* is resized to zero. If *outArray* has not yet been defined, it is created as a text array.

Note: There is no longer any need to copy a globals array just to manipulate it. You should use the simpler syntax:

```
globals{inKey}
```

The resulting value may be used with all of the array commands such as **append to array** and **DELETE ELEMENT**.

get global array size

version 2

get global array size(inKey) → Longint

Parameter	Type	Description
inKey	Text	→ Key of global item to retrieve
Result	Longint	← Size of array

Discussion

This command searches the global collection for the item with the key *inKey*.

If the item is found and its value is an array, the size of the array is returned.

If the item is found and its value is not an array, an error is generated and execution is aborted.

If the item is not found, zero is returned.

Note: This command has been superceded by the simpler syntax:

Size of array(globals{inKey})

get global item

version 2

get global item(inKey) → Longint

Parameter	Type	Description
inKey	Text	→ Key of collection item to retrieve
Result	Longint	← Iterator for collection

Discussion

This command searches the global collection for the item with the key *inKey*.

If the item is found, an iterator reference for the item is returned.

If the item is not found, an empty iterator is returned. For information on empty iterators, see “Iterating Over a Collection” on page 102.

get global keys

version 2

```
get global keys(outKeys)
```

Parameter	Type	Description
outKeys	String/Text Array	← Receives the collection keys

Discussion

This command fills *outKeys* with all of the keys in the global collection. If *outKeys* has not yet been defined, it is created as a string array.

If *outKeys* is defined but is not a string or text array, an error is generated and execution is aborted.

set global

version 2

```
set global(inKey; inValue {; inKeyN; inValueN | inIndex})
```

Parameter	Type	Description
inKey	Text	→ Key of collection item to store
inValue	<any>	→ Value to set for the given item
inIndex	Longint	→ Index of array element to set

Discussion

This command searches the global collection for the item with the key *inKey*.

If the item is found and its value is not an array, you may pass more than one key/value pairs to set multiple items at once. If the item is found and its value is an array, you may pass an index to set an element of the array.

If the item is found and an index is not specified, the item's value is replaced with *inValue*.

If the item is found, its value is an array, and an index is specified, the given array element is set. If the index is out of range or the type of *inValue* is not assignment-compatible with the array, an error is generated and execution is aborted.

If the item is not found and an index is not specified, a new item is added to the collection with the given keys and values.

If the item is not found and an index is specified, an error is generated and execution is aborted.

Note: For scalar values this command has been superseded by the simpler syntax:

```
globals{inKey} := inValue or globals{inKey}{inIndex} := inValue
```

set global array

version 2

```
set global array(inKey; inArray)
```

Parameter	Type	Description
inKey	Text	→ Key of collection item to set
inArray	Array	→ Array to set for the given item

Discussion

This command searches the global collection for the item with the key *inKey*.

If *inArray* is not an array, an error is generated and execution is aborted.

If the item is found, its value is replaced with *inArray*.

If the item is not found, a new item is added to the collection with the given key and array.

Note: This command has been superceded by the simpler syntax:

```
globals{inKey} := inArray
```

count globals

version 2

```
count globals → Longint
```

Parameter	Type	Description
Result	Longint	← Number of items in collection

Discussion

This command returns the number of items in the global collection.

delete global

version 2

delete global(inKey)

Parameter	Type	Description
inKey	Text	→ Key of global item to delete

Discussion

This command searches the global collection for the item with the key *inKey*. To delete more than one item, you may use a wildcard in the key. All items that match will be removed from the collection.

lock globals

version 2

lock globals

Discussion

This command locks the global collection such that no other processes may access it. This is the equivalent of using an interprocess semaphore in 4D.

For information on globals locking, see “Locking and Unlocking the Globals” on page 206.

unlock globals

version 2

unlock globals

Discussion

This command unlocks the global collection so that other processes may access it. This is the equivalent of using an interprocess semaphore in 4D.

For information on globals locking, see “Locking and Unlocking the Globals” on page 206.

Iterators

As with arrays, it is frequently useful or necessary to iterate over the contents of a collection. With arrays this is easy; you just use a **For** loop and index into the array to access a specific element.

Collections, on the other hand, are not necessarily stored in such a way that a numeric index can efficiently be used to access a particular item. Thus it is necessary to provide another mechanism, known as *iteration*. Iteration means that you start at the “beginning” of a collection and advance one item at a time towards the “end” of the collection. At each step the key and value of the current item can be retrieved. This process is known as *iterating over* a collection.

Using for each

There are two ways of iterating over a collection in Active4D. The easiest and most direct way is to use the **for each/end for each** looping control structure. The basic method of using **for each** looks like this:

```
for each(form variables; $key; $value)
    writebr('$key=$value')
end for each
```

In the example above, the collection being iterated over is the *form variables* collection. Within each pass through the loop, *\$key* is set to the key of the current item, and *\$value* is set to the value of the current item. If the item’s value is an array, *\$value* receives a copy of the array.

Note: In previous versions of Active4D, collection keys were kept in alphabetical order, although officially no order was guaranteed. In v6 the keys are kept in no predictable order, so if you have code that relies on alphabetical keys, you will have to get the collection keys, sort them, and then use the sorted keys to access the collection items.

Using Iterators

An iterator is represented by a *Longint* and can be thought of as a special kind of pointer. When an iterator is first created, it points at the first item in a collection. Given an iterator, you can retrieve the key or value of the item it points to and advance to the next item. Of course, you need to know when you have reached the end of the collection, so iterators also provide a way of telling if there are more items left.

The basic template for iterating over a collection looks like this:

```
$it := <get an iterator command>

while (more items($it))
  $key := get item key($it)
  $value := get item value($it)
  next item($it)
end while
```

Pretty simple. You can also test the type of an item's value in case it might be an array, and get the item's array if it is.

Iterator Validity

Active4D tests the validity of the iterator passed to each of the iterator commands. If the iterator is not valid, an error is generated and execution is aborted. Thus you do not have to worry about crashing the server because of a bad iterator.

All iterators become invalid when execution of a script ends.

Warning: If you delete or add an item in a collection, any iterators for that collection will no longer be valid. Thus you should not attempt to delete or add items while iterating over a collection.

for each/end for each

version 3.0

```
for each(inCollectionRef; outKey {; outValue})
```

Parameter	Type	Description
inCollectionRef	Longint	→ Collection handle or iterator
outKey	<any>	← Receives the item key
outValue	<any>	← Receives the item value

Discussion

For a full discussion of **for each/end for each**, see “for each/end for each” on page 245.

more items

version 2

```
more items(inIterator) → Boolean
```

Parameter	Type	Description
inIterator	Longint	→ Iterator
Result	Boolean	← True if more items in collection

Discussion

This command returns *True* if there are more items in the collection to be iterated.

next item

version 2

```
next item(iolterator)
```

Parameter	Type	Description
iolterator	Longint	→ Iterator

Discussion

This command advances the iterator to the next item in the collection.

get item key

version 2

get item key(iterator) → Text

Parameter	Type	Description
iterator	Longint	→ Iterator
Result	Text	← Key of current item

Discussion

This command returns the key of the item currently pointed to by the iterator.

get item value

version 2

get item value(iterator) → <any>

Parameter	Type	Description
iterator	Longint	→ Iterator
Result	<any>	← Value of current item

Discussion

This command returns the value of the item currently pointed to by the iterator. If the value is an array, the current value of the array is returned.

get item type

version 2

get item type(iterator) → Longint

Parameter	Type	Description
iterator	Longint	→ Iterator
Result	Longint	← Type of current item

Discussion

This command returns the type of the item currently pointed to by the iterator.

get item array**version 2**

get item array(inIterator; outArray)

Parameter	Type	Description
inIterator	Longint	→ Iterator
outArray	Array	← Receives a copy of item's array

Discussion

This command returns the array value of the item currently pointed to by the iterator.

If the iterator is not currently pointing at an item with an array value, an error is generated and execution is aborted.

is an iterator**version 2**

is an iterator(inIterator) → Boolean

Parameter	Type	Description
inIterator	<any>	→ Iterator
Result	Boolean	← True if a valid iterator

Discussion

This command returns *True* if *inIterator* is a valid iterator reference. If the value is not a Longint, this command returns *False*.

JSON

If you are using an AJAX-based Javascript library in your web sites, at some point it is likely you will need to create JSON (see json.org) formatted data within Active4D.

Active4D provides a full set of commands for easily creating properly formatted JSON data from all of the data types Active4D supports, including collections. In addition, there are commands for quickly adding a selection of records or a RowSet to JSON data.

Note: The functionality of the a4d.json library has been replaced by native JSON commands, which are *much* faster (up to 50x) and more powerful. For backwards compability, the a4d.json library has been retained, but is now just a thin wrapper around the built in JSON commands, resulting in much greater speed.

It is recommended that you convert any a4d.json-based code to use the built in commands, as the a4d.json library will be deprecated in a future version.

Using the commands

The JSON commands have two primary interfaces: a high level interface and a low level interface. Most of the time you will use the high level interface to create, populate and output a JSON "object". The high level command names begin with "add", "start" or "end". If you need more control over the format of the JSON data, you can use the low level "json encode" command.

UTF-8 and Ajax

When using Ajax requests, posted data is encoded as UTF-8 and UTF-8 encoded data is expected in return.

Active4D handles this transparently. When Active4D detects an Ajax request, it automatically decodes posted UTF-8 data into Unicode. If you use the JSON commands to return data to an Ajax request, the data is automatically encoded as UTF-8.

new json

v6.1

```
new json{{inWrap}} → Object
```

Parameter	Type	Description
inWrap	Boolean	→ True to wrap the JSON data within an enclosing object
Result	JSON Object	← Object to be used with other JSON commands

Discussion

This command creates a new JSON “object” that can be used with other commands in this library to add JSON-formatted data. JSON objects are automatically cleared at the end of the current request.

If *inWrap* is not passed, it defaults to *True*. If *inWrap* is *True*, when **json to text** or **write json** is called the output will be wrapped within an enclosing object ({}).

add to json

v6.1

```
add to json(ioJSON; inKey; inValue {; *; inFilter}; inKeyN; inValueN {; *; inFilterN};)
```

Parameter	Type	Description
ioJSON	JSON object	→ Object to add data to
inKey	Text	→ Key for data item
inValue	<any>	→ Data to add
inFilter	Text	→ Filter for collections

Discussion

This command adds data to a JSON object. You may add an unlimited number of key/value pairs with a single call. All Active4D data types may be used, including arrays and collections.

If *inKey* is not an empty string, a new keyed item is added to the object. If *inKey* is empty, data is added without a key. Ordinarily you would only add data without a key if you are dynamically building an array using the **start json array** and **end json array** commands.

The data in *inValue* is converted to JSON format according to its type:

Value type	Result
String/Text	Encoded string, double-quoted
Number	Number
Date	Date in IETF “Mon Day, Year” format

Value type	Result
Boolean	<i>true</i> or <i>false</i>
Array	Encoded according to element type
Collection	Encoded as a subobject
JSON object	Added as a subobject
Other	<i>null</i>

If *inValue* is a collection and it is followed by *** and *inFilter*, *inFilter* is used to filter the contents of *inValue*. For more information, see “json encode” on page 234.

The array types supported by add to json are those whose scalar elements types are supported. If an element of longint array is a collection handle, the collection is encoded. Arrays of an unsupported type will be added as an empty JSON array.

Examples

To understand how this command works, let’s look at some examples to see the resulting output. First let’s look at the output from the basic data types:

```
$json := new json
add to json($json; "name"; "Sri Chinmoy"; "age"; 76; \\  
    "birthdate"; !08/27/1931!; "Bengali"; true; \\  
    "time"; ?07:13:27?)
write json($json)

// The output is:
{"name":"Sri Chinmoy","age":76,"birthdate":"Aug 27, 1931",  
"Bengali":true,"time":null}
```

Notice that the “time” item returned a value of *null*, because time is not a supported type in JSON. Also notice how we could add multiple key/value pairs at once.

Now let’s look at what happens if we add a collection and an array to a JSON object.

```
$json := new json
$info := new collection("age"; 76; "birthdate"; !08/27/1931!; \\  
    "Bengali"; true; "time"; ?07:13:27?)
array text($siblings; *; "Chitta"; "Mantu"; \\  
    "Hriday"; "Lily"; "Arpita"; "Ahana")
add to json("info"; $info; "siblings"; $siblings)
$json->write

// The output is:
{"name":"Sri Chinmoy","info":{"age":76,"birthdate":"Aug 27,  
1931","Bengali":true,"time":null},"siblings":["Chitta", "Mantu",  
"Hriday", "Lily", "Arpita", "Ahana"]}
```

As you can see, the items in a collection are added as a subobject with the given key. The values in a collection are added with the **add** command.

Finally, let's look at an example of adding a value with no key.

```
array text($siblings; *; "Chitta"; "Mantu"; \\
          "Hriday"; "Lily"; "Arpita"; "Ahana")

$json := new json
add to json($json; "name"; "Sri Chinmoy")
start json array($json; "siblings")

for each($siblings; $name)
  add to json($json; ""; $name)
end for each

end json array($json)
write json($json)

// The output is:
{"name":"Sri Chinmoy","siblings":["Chitta","Mantu","Hriday",
"Lily","Arpita","Ahana"]}
```

add datetime to json

v6.1

add datetime to json(ioJSON; inKey; inDate; inTime; inTimezone)

Parameter	Type	Description
ioJSON	JSON object	→ Object to add data to
inKey	Text	→ Key for data item
inDate	Date	→ Date portion of datetime
inTime	Time	→ Time portion of datetime
inTimezone	Number	→ Timezone portion of datetime

Discussion

This command adds an IETF format datetime item to a JSON object. Such an item can be turned into a Javascript Date object by passing the item string to the Date constructor.

If *inKey* is not an empty string, a new keyed item is added to the object. If *inKey* is empty, a datetime item is added without a key. Ordinarily you would only add a datetime without a key if you are dynamically building an array using the **start json array** and **end json array** commands.

inTimezone should be in the timezone indicated in the *inTimezone* parameter. *inTimezone* should be minute offset from GMT.

Example

```

$json := new json
// add a datetime in EST (GMT-5)
add datetime to json($json; "DA"; !02/17/1980!; ?20:31:07?; \\
-5 * 60)

write json($json)

// The output is:
{"DA":"Feb 17, 1980 20:31:07 GMT-0500"}

// Javascript on the client, JSON is in a variable called json
var da = new Date(json.DA)

```

add function to json

v6.1

add function to json(ioJSON; inName; inBody)

Parameter	Type	Description
ioJSON	JSON object	→ Object to add data to
inName	Text	→ Function name
inBody	Text	→ Function body

Discussion

This command adds an item to *ioJSON* with the key *inName* and a value of *inBody* as is, unquoted and unencoded. This is useful when you need to add a function (such as a handler or callback) to your JSON data.

Example

```

$json := new json
add function to json($json; "renderer"; \\
""function(value, metadata, record) {
  return record.data.title + " " + value;
}""")

```

add rowset to json

v6.1

```
add rowset to json(ioJSON; inRowSet {; inCountKey {; inDataKey {; inMap
{; inFirst {; inLimit}}}}})
```

Parameter	Type	Description
ioJSON	JSON object	→ Object to add data to
inRowSet	RowSet	→ RowSet from which to get data
inCountKey	Text	→ Key for row count item
inDataKey	Text	→ Key for row data item
inMap	Text	→ JSON name to RowSet column map
inFirst	Number	→ Index of first row to add
inLimit	Number	→ Maximum number of rows to add

Discussion

This command adds rows from *inRowSet* to the JSON object *ioJSON*. If you have a RowSet and you want to use it for generating JSON data, this command is the fastest and easiest way to do so. If the RowSet is selection-based and is not being used for other purposes, in most cases you will want to use **add selection to json** instead of this command.

The RowSet's rows are added as an array of objects, with each object containing one item for each column of data. The column data is converted as if were passed to the **add to json** command. If *inDataKey* is passed and is non-empty, the array will have the key *inDataKey*.

If *inCountKey* is passed and is non-empty, an item will be added to *ioJSON* whose key is *inCountKey* and whose value is the number of rows in *inRowSet*.

Note: The number of rows returned with *inCountKey* is the result of calling *\$inRowSet->rowCount*, not *\$inRowSet->sourceRowCount*.

If *inMap* is not passed or is empty, all of the columns in *inRowSet* will be added to the row array. If *inMap* is passed and is non-empty, it must be a semicolon-delimited list of mappings from RowSet column names to JSON key names. If the RowSet column name will be used as is, it is sufficient to use just the column name. If you want to rename a RowSet column, then the mapping should be a *<JSON key>:<RowSet column>* pair. This allows you to specify a subset of the RowSet columns for inclusion in the JSON data, and/or to rename the RowSet columns.

If *inFirst* is passed and is ≥ 1 , it specifies the one-based index of the first row from *inRowSet* that will be added to *self*. If *inLimit* is passed and is ≥ 0 , it specifies the maximum number of rows from *inRowSet* that will be added to *self*. Together, *inFirst* and

inLimit make it easy to specify a subset of rows, which is typically the case when paging through a large RowSet.

Note: Depending on the Ajax toolkit you are using, it is likely that the start index for a page of data will be zero-based. It is up to you to add 1 to make it one-based before using the value for *inFirst*.

Examples

Let's look at a few examples to illustrate the typical use of the various options. First, we'll create a RowSet and then add it a JSON object.

```
// Assume we have a selection of 3 [employee] records
$map := ""
name: `concat(" "; [employee]first; [employee]last)`;
id: [employee]id;
birthdate: [employee]birthdate""
$rs := RowSet.newFromSelection(->[employee]; $map)
$json := new json
add rowset to json($json; $rs; "count"; "rows")
write json($json)

// The output is:
{"count":3,"rows":[{"name":"Tiny Tim","id":31,"birthdate":"Apr
12, 1932"}, {"name":"James Taylor","id":27,"birthdate":"Mar 12,
1948"}, {"name":"Pat Metheny","id":13,"birthdate":"Aug 12,
1954"}]}
```

Now let's add the rest of the parameters to see the effect. This time we want to eliminate the *id* column from the JSON data, and we want to rename the RowSet *birthdate* column to the JSON key *dob*. In addition, we are receiving the starting index in a query parameter called "start" whose value is "2", and the number of rows to return in a query parameter called "size" whose value is "1".

```
// RowSet setup is the same as the example above

// Use "name" column as is, rename "birthdate" to "dob"
$jmap := "name;dob:birthdate"

// If "start" query param is not passed, default to "0",
// convert it to one-based number
$first := num($attributes{"start"} | "0") + 1

// If "size" query param is not passed, default to "20"
$limit := num($attributes{"size"} | "20")
$json := new json
add rowset to json($json; $rs; "count"; "rows"; $jmap; \
    $first; $limit)
write json($json)

// The output is:
{"count":3,"rows":[{"name":"Pat Metheny","dob":"Aug 12, 1954"}]}
```

Note that the *count* item still returns 3, because the number of rows in the RowSet is still 3, even though there is only one row in the *rows* array. The reason for this is because when you are showing paging information for a RowSet, you usually want to display something like “Displaying records <start>-<end> of <total>”, so you always need the total number of rows in the source dataset.

add selection to json

v6.1

```
add selection to json(ioJSON; inTable {}; inCountKey {}; inDataKey {}; inMap
  {}; inFirst {}; inLimit}}}})
```

Parameter	Type	Description
ioJSON	JSON object	→ Object to add data to
inTable	Table	→ Main table from which to get data
inCountKey	Text	→ Key for row count item
inDataKey	Text	→ Key for row data item
inMap	Collection	→ JSON name to value map
inFirst	Number	→ Index of first row to add
inLimit	Number	→ Maximum number of rows to add

Discussion

This command adds record data from the current selection of *inTable* to the JSON object *ioJSON*. If you have a selection of records and you want to use it for generating JSON data, this command is the fastest and easiest way to do so.

This command operates in two modes:

- **selection mode:** If *inFirst* # -1, the selection's records are output as an array of objects, with each object containing one item for each field.

inFirst specifies the one-based index of the first record in the current selection of *inTable* that will be added to *self*. If *inLimit* is passed and is ≥ 0 , it specifies the maximum number of records from the current selection of *inTable* that will be added to *self*. Together, *inFirst* and *inLimit* make it easy to specify a subset of records, which is typically the case when paging through a large selection.

Note: Depending on the Ajax toolkit you are using, it is likely that the start index for a page of data will be zero-based. It is up to you to add 1 to make it one-based before using the value for *inFirst*.

- **current record mode:** If *inFirst* = -1, only the current record of the selection is output as a single object (not within an array) with one item for each field.

The field data is converted as if were passed to the **add to json** command. If *inDataKey* is passed and is non-empty, the record data will have the key *inDataKey*.

If *inCountKey* is passed and is non-empty, an item will be added to *ioJSON* whose key is *inCountKey* and whose value is the number of records in the current selection of *inTable*.

If *inMap* is not passed or is empty, all of the fields in *inTable* will be added to the record array, with the field name being the JSON column key. If *inMap* is passed and is non-zero, it must be a collection which maps JSON column keys to column values. The keys in *inMap* are used as the JSON column keys. The values may be one of three types:

- **Table pointer:** If the value is a table pointer, the current record number for *inTable* is output as a JSON number.
- **Field pointer:** If the value is a field pointer, the value of the field is output as if it were passed to the **add to json** command. Fields from tables other than *inTable* may be used if there is a many to one relation (it need not be automatic) between *inTable* and the foreign field's table. If any foreign fields are used in *inMap*, **RELATE ONE**(\$*inTable*->) is executed before each record is output to ensure related data is available.
- **Text:** If the value is text, it must be an expression than returns a value (although using **return** is not necessary). The resulting value is output as if it were passed to the **add to json** command. If you want the result to appended verbatim, without being JSON encoded, prefix the expression with "@". If you want to evaluate the expression in 4D instead of Active4D, prefix the expression (after "@" with "!".

If *inMap* contains an item whose key is "a4d.json.callback", then the value should be a text block of code to execute after each record of *inTable* is loaded. The code does not have to return a value. If you want the code to be executed in 4D instead of Active4D, prefix it with "!". Code executed in Active4D may consist of multiple statements separated by carriage returns. Code executed in 4D may only be one line.

Examples

Let's look at a few examples to illustrate the typical use of the various options. First, we'll create a selection of records and then add it a JSON object.

```
// [employee] table
// id          Longint
// firstname   Alpha20
// lastname    Alpha20
// company_id  Longint, relate one with [company]id
// dob         Date
query([employee]; [employee]contact_id = $attributes{"id"})
$json := new json
add selection to json($json; [employee]; "count"; "rows")
write json($json)

// The output is:
{"count":3,"rows":[{"id":31,"firstname":"Tiny","lastname":"Tim",
"company_id":101,"dob":"Apr 12, 1932"},{"id":27,
"firstname":"James","lastname":"Taylor","company_id":107,
"dob":"Mar 12, 1948"},{"id":13,"firstname":"Pat",
"lastname":"Metheny","company_id":107,"dob":"Aug 12, 1954"}]}
```

Notice how all of the fields were automatically included in the JSON output. Now let's add the rest of the parameters to see the effect.

To make the output more useful, we would like to do the following:

- Include the record number of the [employee] table

- Concatenate the first name and last name into a single name column
- Return the company name instead of its id
- Rename "dob" to "birthdate"

Note: For an example of how to use the *inFirst* and *inLimit* parameters, see the documentation for "add rowset to json" on page 224.

We accomplish this by creating a map and passing it to **add selection to json**:

```
// Selection setup is the same as the example above

$map := new collection

// output record number
$map{"recnum"} := ->[employee]

// output concatenation of firstname + " " + lastname
$map{"name"} := "concat(\" \"; [employee]firstname;
[employee]lastname)"

// output foreign related one field
$map{"company"} := ->[company]name

// rename a field
$map{"birthdate"} := ->[employee]dob

// use a callback
$map{"a4d.json.callback"} := \\
"query([family]; [family]employee_id = [employee]id)\r" + \\
"query([family]; & [family]type = 1)" // type 1 is child

$map{"num_children"} := "records in selection([family])"

$json := new json
add selection to json($json; [employee]; "count"; "rows"; $map)
write json($json)

// The output is:
{"count":3,"rows":[{"recnum":207,"name":"Tiny Tim",
"company":"Tulips, Inc.,"birthdate":"Apr 12, 1932",
"num_children":0},{ "recnum":331,"name":"James Taylor",
"company":"Gorilla Corp.,"birthdate":"Mar 12, 1948",
"num_children":2}, {"recnum":713,"name":"Pat Metheny",
"company":"The Way Up","birthdate":"Aug 12, 1954",
"num_children":1}]}
```

Finally, suppose we have vendors and products they sell, and we want one JSON object for each vendor which contains an array of objects for each product. We could manually build the data using **start json array/start json object** and **end json object/end json**

array. But there is a faster and easier way, using a nested JSON object return by a method that is executed with every vendor record.

```
// Define a method that will output an array of products
// for the current vendor.

method "vendorProducts"
  relate many([vendors]id) // get related products
  $json := new json(false) // don't wrap result in an object
  $map := new collection( \\
    "id"; ->[products]id; "name"; ->[products]name; \\
    "price": ->[products]price)
  add selection to json($json; [products]; "; "; $map)
  return (json to text($json))
end method

// Now encode the vendors

all records([vendors])
$json := new json
$map := new collection( \\
  "id"; ->[vendors]id; "name"; ->[vendors]name; \\
  "products": "@vendorProducts")

// Note we prefixed the method name with "@" so the result
// will not be encoded as a string but appended verbatim.

add selection to json($json; [vendors]; "count"; "rows"; $map)
write json($json)

// Output:
{"count":2,"rows":[{"id":100,"name":"Spacely Sprockets",
"products":[{"id":7,"name":"Widget 100","price":13.99},
{"id":13,"name":"Sprocket","price":7.5}]},{"id":103,
"name":"Cogswell Cogs","products":[{"id":31,"name":"Flange
X","price":27.5}, {"id":47,"name":"Cog","price":3.99}]}}}
```

start json array

v6.1

```
start json array(ioJSON {; inKey})
```

Parameter	Type	Description
ioJSON	JSON object	→ Object to add data to
inKey	Text	→ Key for array item

Discussion

This command adds an array start marker to a JSON object. Ordinarily you would only call this command if you are dynamically building an array using the **start json array** and **end json array** commands instead of using the **add to json** command.

Warning: To ensure valid JSON data, be sure to balance a call to this command with a call to **end json array**.

If *inKey* is not passed or is an empty string, the array start marker is preceded by an item key.

Example

```
array text($siblings; *; "Chitta"; "Mantu"; \\  
          "Hriday"; "Lily"; "Arpita"; "Ahana")  
  
$json := new json  
start json array($json; "siblings")  
  
for each($siblings; $name)  
  add to json($json; ""; $name)  
end for each  
  
end json array($json)  
write json($json)  
  
// The output is:  
{"siblings":["Chitta","Mantu","Hriday", "Lily","Arpita","Ahana"]}
```

end json array

v6.1

```
end json array(ioJSON)
```

Parameter	Type	Description
ioJSON	JSON object	→ Object to add data to

Discussion

This command adds an array end marker to a JSON object. Ordinarily you would only call this command if you are dynamically building an array using the **start json array** and **end json array** commands instead of using the **add to json** command.

Warning: To ensure valid JSON data, be sure to balance a call to this command with a previous call to **start json array**.

Example

See “start json array” on page 230.

start json object

v6.1

```
start json object(ioJSON {; inKey})
```

Parameter	Type	Description
ioJSON	JSON object	→ Object to add data to
inKey	Text	→ Key for subobject

Discussion

This command adds an object start marker to a JSON object. Ordinarily you would only call this command if you are dynamically building an object using the **start json object** and **end json object** commands instead of using the **add to json** command.

Warning: To ensure valid JSON data, be sure to balance a call to this command with a call to **end json object**.

If *inKey* is not passed or is an empty string, the object start marker is preceded by an item key.

Example

```
$json := new json
start json array($json; "employees")

for each([employee])
  start json object($json)
  add to json($json; "name"; [employee]name); \\
  "age"; [employee]age)
end json object($json)
end for each

end json array($json)
write json($json)

// The output is:
{"employees": [{"name": "Tom", "age": 31}, {"name": "Dick", "age": 27}, {"name": "Harry", "age": 42}]}
```

end json object

v6.1

```
end json object(ioJSON)
```

Parameter	Type	Description
ioJSON	JSON object	→ Object to add data to

Discussion

This command adds an object end marker to a JSON object. Ordinarily you would only call this command if you are dynamically building an object using the **start json object** and **end json object** commands instead of using the **add to json** command.

Warning: To ensure valid JSON data, be sure to balance a call to this command with a previous call to **start json object**.

Example

See “start json object” on page 231.

json to text

v6.1

json to text(inJSON) → Text

Parameter	Type	Description
inJSON	JSON object	→ Object from which to get JSON data
Result	Text	← JSON-formatted data

Discussion

This command returns the data that has been added to *inJSON* as JSON-formatted text.

Usually you will not need to call this command directly, since you will want to write the result of this command to the response buffer, and the **write json** command does that for you.

Warning: If you do decide to write JSON data to the response buffer yourself (instead of using **write json**), be sure to use the **write raw** command to prevent any HTML encoding.

write json

v6.1

write json(inJSON; inSetContentType)

Parameter	Type	Description
inJSON	JSON object	→ Object from which to write JSON data
inSetContentType	Boolean	→ True to set the content type

Discussion

This command writes the data that has been added to *self* as JSON-formatted text to the response buffer. This is the primary command you will use to return JSON data to a client.

If *inSetContentType* is not passed or is *True*, the content type of the response is set to "application/json".

write jsonp

v6.1

`write jsonp(inJSON)`

Parameter	Type	Description
<code>inJSON</code>	JSON object	→ Object from which to write JSON data
<code>inCallback</code>	String	→ Callback expected by JSONP request
<code>inSetContentType</code>	Boolean	→ True to set the content type

Discussion

This command writes the data that has been added to *inJSON* as JSON-formatted text to the response buffer, wrapped in a call to the function *inCallback*. This is the primary command you will use to return JSON data to a client that is using the JSONP protocol.

If *inSetContentType* is not passed or is *True*, the content type of the response is set to "application/json".

Example

Assume your front end makes a JSONP request with the following URL:

```
/enrollments/list?cb=CPJSONPConnectionCallbacks.callback32626
```

You build the JSON response and then return it like this:

```
$json := new json
// add data to $json
$callback := $attributes{"cb"} // assuming fusebox
write jsonp($json; $callback)
```

json encode

v6.1

`json encode(inValue {; inFilter}) → Text`

Parameter	Type	Description
<code>inValue</code>	<any>	→ Data to JSON encode
<code>inFilter</code>	Text	→ Filter for collections
<code>Result</code>	Text	← Encoded value

Discussion

This low level command encodes data into JSON format. All Active4D data types may be used, including arrays and collections. Usually you would not use this command directly, but use the high level **add to json** command instead.

When encoding a collection, this command encodes the data in *inValue* as a JSON object, with each key/value pair in the collection becoming a key/value pair in the JSON object. If the value has no corresponding JSON type, the value *null* is returned.

If *inFilter* is passed and is non-empty, it is used as a matching expression to determine which items from *inCollection* are encoded. The rules for the filter expression are as follows:

- If the filter begins with "#", it performs an exclusion, i.e. all items that match the filter are excluded.
- Otherwise only items that match the filter are included.
- If the first character after the optional "#" and the last character is "/", it is considered a regular expression pattern and regex matching is done.
- Otherwise simple string comparison is performed.

Examples

Let's take a look at some simple filters to see how they affect the output of this command.

```
$c := new collection("foo"; 7; "bar"; !04/13/1964!; "baz"; false)

// simple string matching
$enc := json encode($c; "foo")
// $enc = {"foo":7}

// simple string matching with wildcard
$enc := json encode($c; "b@")
// $enc = {"bar":"Apr 13, 1964","baz":false}

// exclusion matching, simple string
$enc := json encode($c; "#bar")
// $enc = {"foo":7,"baz":false}

// exclusion matching, simple string with wildcard
$enc := json encode($c; "#b@")
// $enc = {"foo":7}

// regex matching
$enc := json encode($c; "/foo|bar/")
// $enc = {"foo":7,"bar":"Apr 13, 1964"}

// exclusion matching, regex
$enc := json encode($c; "#/foo|bar/")
// $enc = {"baz":false}
```

parse json

v6.0r2

```
parse json(inJSON {; inDateKeys {; inThrowOnError{}}) → <any>
```

Parameter	Type	Description
inJSON	Text	→ Text to parse
inDateKeys	Text	→ Regular expression to match keys for date conversion
inThrowOnError	Boolean	→ True to throw on malformed JSON
Result	<any>	← Parsed JSON data

Discussion

This method parses JSON text into its corresponding Active4D value.

All valid JSON types and syntaxes are supported, with the exception that array elements must all be of the same type. JSON types map to Active4D types as follows:

JSON type	Active4D type
object	collection
array	array
string	text
number	real
true/false	boolean
null	nil pointer

Note that JSON has no native representation for dates. To convert textual representations of dates into 4D dates, pass a delimited regular expression in *inDateKeys*. JSON strings or string arrays whose keys match the regular expression will be considered for conversion. Conversion occurs if the string value matches one of the following patterns:

Pattern	Comments
Mmm d, yyyy	IETF date format. Mmm is a 3-letter English month abbreviation.
yyyy-mm-dd	ISO Date format
yyyy-mm-ddThh:mm:ss	

Also note that all collections created from JSON objects will be local.

Note: Because this command may return an array, you should always use the super assign operator (`::=`) to assign the result of this command.

If *inThrowOnError* is True (the default), malformed JSON will throw an error. Otherwise it will skip the item and continue if it can recover. If it cannot recover, an error will be thrown.

Example

```
// Given the JSON data:  
{ "name": "Sara", "birthdate": "Dec 10, 1931" }  
  
// When parsing, we can convert "birthdate" to a 4D date:  
$data := parse json($json; "/^birthdate$/"; true)
```

Language

Active4D supports every major feature of the 4D language. In addition, it extends the language to make code writing more efficient.

You should especially learn to use the **choose** command, as it is tremendously useful.

call 4d method

version 1

```
call 4d method(inMethodName {;inParam1 {...;inParamN}}) → <any>
```

Parameter	Type	Description
inMethodName	Text	→ Name of method to call
inParam1..N	<any>	→ Method parameters
Result	<any>	← Value returned by method

Discussion

This command executes the given 4D method, passing any parameters which follow the method name. It is critical that the parameters you pass are compatible with the parameter declaration in the 4D method. If the method returns a value, it may be ignored.

Note: The primary purpose of this command is to allow indirect method calls. Under normal circumstances this command is unnecessary, as you can call 4D methods directly with the same syntax as you would in 4D.

If *inMethodName* does not specify an existing 4D method, an error is generated and execution is aborted.

For more on the purpose and use of this command, see “Indirect Method Calls (aka Poor Man's method pointers)” on page 99.

call method

version 1

```
call method(inMethodName {;inParam1 {...;inParamN}}) → <any>
```

Parameter	Type	Description
inMethodName	Text	→ Name of method to call
inParam1..N	<any>	→ Method parameters
Result	<any>	← Value returned by method

Discussion

This command executes the given Active4D method, passing any parameters which follow the method name. If the method returns a value, it may be ignored.

Note: The primary purpose of this command is to allow indirect method calls. Under normal circumstances this command is unnecessary, as you can call Active4D methods directly with the same syntax as you would a 4D method.

You may specify a library method by using the *library.method* syntax, just as you would if you were directly calling the method. If *inMethodName* does not specify an existing Active4D method, an error is generated and execution is aborted.

Normal rules for parameter passing to Active4D methods apply to this command. For more information on calling Active4D methods, see Chapter 8, “Methods.”

choose

version 2

choose(inCondition; trueExpression; falseExpression) → <any>

Parameter	Type	Description
inCondition	Boolean	→ Determines which expression to evaluate
trueExpression	<any>	→ Expression to evaluate if inCondition is True
falseExpression	<any>	→ Expression to evaluate if inCondition is False
Result	<any>	← trueExpression or falseExpression

Discussion

This command evaluates the Boolean expression *inCondition*. If the expression evaluates to *True*, the command evaluates and returns the result of *trueExpression*. If the expression evaluates to *False*, the command evaluates and returns the result of *falseExpression*.

This command is basically a way of performing an inline **If/Else/End if**. Used properly, it can greatly streamline your code.

For example, suppose you are building a table with a list of records. If a table cell has no data at all, some browsers will not display the cell as empty, but rather as non-existent. To prevent this, you insert a non-breaking space () in the cell.

Without the use of **choose**, your code would look something like this:

```
<table>
  <%
    for ($i; 1; records in selection([contacts]))
      goto selected record([contacts]; $i)
  %>
  <tr>
    <td>
      <%
        if (length([contacts]name) = 0)
          write("&nbsp;")
        else
          write([contacts]name)
        end if
      %>
    </td>
  </tr>
  <% end for %>
</table>
```

Using **choose**, the **If/Else/End if** block becomes this:

```
<% =choose(length([contacts]name) = 0; "&nbsp;"; [contacts]name)
%>
```

Note that we have streamlined even further by using the = operator as a synonym for **write**. For more information on this use of =, see "Response Buffer" on page 300.

Here's another example where **choose** comes in handy: setting radio buttons. To set a radio button in HTML, you must add the word "checked" to the radio button's tag. Ordinarily, this would require a separate test for each radio button declaration, like this:

```
<input type="radio" name="f_radio" value="1"
  <% if ([contact]rating = 1)
    write("checked")
  end if
%>
/>Good
<br />
<input type="radio" name="f_radio" value="2"
  <% if ([contact]rating = 2)
    write("checked")
  end if
%>
/>Better
<br />
<input type="radio" name="f_radio" value="3"
  <% if ([contact]rating = 3)
    write("checked")
  end if
%>
/>Best<br />
```

Yuck. This would probably lead you to write a library method something like this:

```
method "makeRadio"($inName; $inValue; $inChecked)
  write('<input type="radio" name="$inName" value ="inValue"')

  if ($inChecked)
    write(" checked")
  end if

  write(" />")
end method
```

This is better in one sense, but worse in another; because you have embedded the radio button HTML in a method, you lose the ability to view and edit the checkbox in an HTML editor. Remember, the whole point of Active4D is to embed your code in HTML, not the other way around.

Using **choose**, you can have the best of both worlds, like this:

```
<input type="radio" name="f_radio" value="1"
  <%=choose([contact]rating = 1; "checked"; " ")%> />Good
<br />
<input type="radio" name="f_radio" value="1"
  <%=choose([contact]rating = 2; "checked"; " ")%> />Better
<br />
<input type="radio" name="f_radio" value="1"
  <%=choose([contact]rating = 2; "checked"; " ")%> />Best
```

define

version 1

define(inName; inValue)

Parameter	Type	Description
inName	Identifier	→ A name for the constant
inValue	<any>	→ The value to assign to the constant

Discussion

This keyword defines a new named constant. The name must conform to the rules for 4D process variables. The value can be any valid expression.

Named constants have scope, like local variables. Constants defined outside of a library are global in scope, and may be accessed anywhere, including within the body of methods.

Constants defined within libraries are scoped to the library, like library methods. Library constants are accessible to all methods in the library without using the library name. Outside of the library, library constants are accessible by using `<library>.<constant>` notation, where `<library>` is the library name and `<constant>` is the constant name, or

simply by using `<constant>` if its name does not clash with any other name in the current scope.

If used within a library definition, the **define** command must appear after between the **library** and **end library** keywords, but outside of any method definitions. It is an error to define a constant within the body of a method.

If a library constant referred to within a library method has the same name as a global constant, the library constant is used. However, if you refer to a constant outside of a library method that is defined in more than one imported library, or defined in an imported library and the global scope, an error is generated because Active4D cannot disambiguate the constant reference.

Thus, if a global constant has the same name as an imported library constant, it will no longer be accessible.

The **define** command is useful anywhere you would normally want to use a named constant within 4D, but you have not created one. For example, suppose you have a Longint field called `[Sales_Leads]Rating` which classifies a sales lead according to the following values:

Value	Description
1	Cold
2	Warm
3	Hot
4	Nuclear

If you have not defined 4DK# named constants for these values, it would be nice to refer to them by names instead of numbers. You could create a file called "constants.a4d" which contained the following code:

```
<%
  define(kSalesLead_Cold; 1)
  define(kSalesLead_Warm; 2)
  define(kSalesLead_Hot; 3)
  define(kSalesLead_Nuclear; 4)
%>
```

Within several scripts you need to use those constants. You cannot use **include**, because it is an error to **define** the same constant more than once. This is where **require** comes in. By using **require** before the code that uses the constants, like this:

```
<%
  require("constants.a4d")
  query([sales_leads];[sales_leads]rating = kSalesLead_Nuclear)
  // and so on
%>
```

you can ensure they are defined without worrying about whether or not they have already been defined within the current script execution.

EXECUTE**version 2 (enhanced 4D)**

EXECUTE(inText {; *}) → <any>

Parameter	Type	Description
inText	Text	→ Text to execute
*	*	→ Execute as embedded code
Result	<any>	← Optional return value

Discussion

This command can be used like the standard **EXECUTE** command in 4D, which is limited to executing a single line of code.

Active4D extends the **EXECUTE** command to allow execute of an entire block of code, including control structures like loops and **If/Else/End if**. This allows you to use techniques such as storing scripts in the database to prevent manipulation by the end user, like this:

```
query([scripts];[scripts]name="query customer by name")
execute([scripts]text)
```

Active4D also extends **EXECUTE** by allowing you to return a value using the return keyword within the executed text, like this:

```
$text := "return (size of array(%s))" % $arrayName
$size := execute($text)
```

If the option * argument is passed, *inText* is treated as HTML with embedded Active4D code instead of raw Active4D code. Effectively this is the same as an **include**, but the code to include is passed directly into the command.

execute in 4d**version 4.0**

execute in 4d(inText {; *})

Parameter	Type	Description
inText	Text	→ Text to execute
*	*	→ Expect a result

Discussion

This command executes *inCode* within 4D, and is subject to all of the restrictions of 4D's **EXECUTE** command.

If `*` is passed, the result of the executed code can be returned as the result of the command. If there is an error in the parsing or execution of `inCode`, Active4D throws a syntax error.

In conjunction with the ability to create process/interprocess variables within Active4D, this command relieves you of the need to write wrapper 4D methods to implement commands that are not supported natively within Active4D.

For example, let's assume you need to use 4D's **Square root** command. Instead of writing a method to call the command, you can do this:

```
$num := 7
$root := execute in 4d("Square root(%d)" % ($num); *)
```

for each/end for each

version 3.0

```
for each(inCollection; outKey {; outValue})
for each(inArray; outValue {; outIndex})
for each(inText; outCharacter {; outIndex})
for each(inTable {; outIndex})
```

Parameter	Type	Description
inCollectionRef	Longint	→ Collection handle or iterator
outKey	<any>	← Receives the item key
outValue	<any>	← Receives the item value
OR		
inArray	Array	→ Array to iterate over
outValue	<any>	← Receives the current array element
outIndex	Number	← Receives the current array index
OR		
inText	Text	→ Text to iterator over
outCharacter	<any>	← Receives the current character
outIndex	Number	← Receives the current character index
OR		
inTable	Table	→ Table to iterator over
outIndex	Number	← Receives the selected record index

Discussion

This control structure is a generalized sequence iterator, where a sequence is defined as a sequence of values, including collections, arrays, strings and selections. Depending on the sequence which is passed as the first argument, **for each** takes different arguments.

Each **for each** must be balanced with a corresponding **end for each**.

Collections

If a collection reference is passed as the first argument, **for each** iterates over the items in the collection. In the body of the loop, *outKey* contains the key of the current collection item, and *outValue* (if passed) contains a copy of the current collection item's value. If the item value is an array, *outValue* receives a copy of the array.

Example

```
for each(form variables; $key; $value)
  writebr($key + "=" + $value)
end for each

$c := new collection
$c{"name"} := "Dave"
$c{"age"} := 31

for each($c; $key)
  writebr('$key=$c{$key}')
end for each
```

Arrays

If an array is passed as the first argument, **for each** iterates over the elements in the array. In the body of the loop, *outValue* contains the current array element, and *outIndex* (if passed) contains the index of the element.

Example

```
for each($myArray; $value; $i)
  writebr('$i: $value')
end for each

// above code is equivalent to:
for($i; 1; size of array($myArray))
  writebr('$i: $myArray{$i}')
end for
```

Strings

If text is passed as the first argument, **for each** iterates over the characters in the text. In the body of the loop, *outCharacter* contains the current character, and *outIndex* (if passed) contains the index of the character.

Example

```

$s := "foo"

for each($s; $char; $i)
  writebr('[[ $i ]]: $char')
end for each

// above code is equivalent to:
for($i; 1; length($s))
  writebr('[[ $i ]]: $s[[ $i ]]')
end for

```

Selections

If a table reference is passed as the first argument, **for each** iterates over the records in the current selection of the table. In the body of the loop, *outIndex* (if passed) contains the selected record index.

Example

```

for each([contacts]; $i)
  writebr('$i. [contacts]fullname')
end for each

// above code is equivalent to:
first record([contacts])

while(not(end selection([contacts])))
  writebr("%d. %s" % (selected record number([contacts]); \\  

    [contacts]fullname))
end while

```

Get pointer

version 1
modified v5

Get pointer(inName) → Pointer

Parameter	Type	Description
inName	Text	→ Name of variable
Result	Pointer	← Pointer to variable

Discussion

This command functions exactly like the **Get pointer** command in 4D, with the addition that you may use "<>" with an interprocess variable name on macOS.

get throw code

v6.0

get throw code → Longint

Parameter	Type	Description
Result	Longint	← Error code used in throw command

Discussion

This command returns the error code passed to the most recent **throw** command.

If no error code was passed to **throw** or this command is not executed within an error handler (**in error** is True), it returns zero.

get throw message

v6.0

get throw message → Text

Parameter	Type	Description
Result	Text	← Error message used in throw command

Discussion

This command returns the error message passed to the most recent **throw** command.

If no error message was passed to **throw** or this command is not executed within an error handler (**in error** is True), it returns an empty string.

global

version 1

global(localVar {; localVarN} | *)

Parameter	Type	Description
localVar *	Local Variable *	→ Local variable to bring into current scope, or all locals

Discussion

When you call an Active4D method, the method has its own local variables in a separate scope, just as in 4D.

If you want to access or modify local variables outside of the method, ordinarily you would pass the variables to the method as by-value or by-reference parameters.

However, there are times when it is necessary to have direct access to local variables outside of the method.

The action of this command depends on where it is used and whether or not the named variables already exist.

- If this command is used in the global scope, it does nothing.
- If this command is used within a method and *localVar* exists in the global scope, it will be accessible within the scope of the method.
- If this command is used within a method and *localVar* does not exist in the global scope, if it is defined within the method it will be in the global scope.
- If this command is used within a method and you pass * instead of one or more variable names, *all* locals in the global scope will be accessible within the method and all locals defined within the method will be in the global scope.

Variables created with **set local** are also affected by the use of this command.

In general you should avoid using the form **global(*)**, because you are likely to create variables in the global scope unintentionally.

Example

In this example, the variable *\$foo* is defined in the global scope before calling the method *foobar*. Because *\$foo* is declared as global within *foobar*, it is accessible within that method. Because *\$bar* is declared global before being defined, it is then accessible in the global scope outside of *foobar*, but not until after *foobar* executes.

```
method "foobar"
  global($foo; $bar)
  writebr($foo)
end method

$foo := "foo"
foobar
writebr($bar) // this works because it is global
```

import

version 1
modified version 4.0

```
import(inLibName {; *}) { → Boolean }
```

Parameter	Type	Description
inLibName	Text	→ Name of library to load
Result	Boolean	← True if successfully imported

Discussion

This command loads an Active4D library. For a full discussion of library importing, see "Importing Libraries" on page 118.

If the optional `*` is passed, no error is generated if the library is not successfully imported, and a Boolean value is returned to indicate the success of the import. This allows you to attempt to import optional libraries.

include

version 1

include(inPath)

Parameter	Type	Description
inPath	Text	→ Path of file to include

Discussion

This command includes the file specified by *inPath*. The path may be absolute or relative. An absolute path will be relative to the web root directory, a relative path will be relative to the currently executing file.

To include a file outside of the web root, you must either create an alias within the web root to the include file or the directory in which the include file lives, or use relative motion within the include path. Here are some examples:

```
// Here we are using relative motion in the include path.
// Note that ".inc" does NOT have to be registered as an
// executable extension, because the include command doesn't
// check that.
include("../includes/foobar.inc")

// Here we have created an alias called "includes" within the
// web root which points to a directory outside the web root.
include("/includes/foobar.inc")
```

For more information on includes, see “Including Other Files” on page 97.

include into

version 2

include into(inPath; outBuffer)

Parameter	Type	Description
inPath	Text	→ Path of file to include
outBuffer	Text/BLOB	← Temporarily becomes the response buffer

Discussion

This command is identical to the **include** command, but it temporarily makes *outBuffer* the response buffer for the duration of the **include** (and any nested includes). This is

extremely useful when you want to create dynamic output that will go somewhere other than the browser.

For example, if you want to generate an invoice and email it to a client, you could do the following:

- 1 Create a nicely formatted HTML invoice, embedding Active4D code to fill in the details of the invoice.
- 2 Use **include into** to dynamically generate the invoice and place the result in a text variable.
- 3 Call **a4d.utils.sendMail** to email the generated invoice as the email body.

outBuffer may be a scalar (non-array) variable, an element of a string or text array, or a reference to a collection item. If not, an error is generated and execution is aborted.

If an error occurs in the code of the included file (or any nested includes), *outBuffer* is left untouched and the error message is placed in the normal response buffer for display in the browser.

longint to time

version 1

longint to time(inLong) → Time

Parameter	Type	Description
inLong	Longint	→ Value to convert to a time
Result	Time	← Converted value

Discussion

This command converts a Longint value into a Time value.

Unless you use a compiler declaration, Active4D's variables are variant in their type. To assign a Longint to a variant variable as a Time you must use this command. This is specially useful when you are storing Times in a Longint array and you want to retrieve the elements as Times.

Note: You could also use **C_TIME** to fix a variable's type to Time and assign a Longint to it. This would do type conversion for you.

method exists

version 4.0

method exists(inMethodName {; outLibName}) → Boolean

Parameter	Type	Description
inMethodName	Text	→ Name of method to check
outLibName	Text	← Name of method's library
Result	Boolean	← True if method exists

Discussion

This command determines if the method with the name *inMethodName* exists. If *inMethodName* is a fully qualified *library.method* name, the library is imported if necessary and then that library is checked for the given method name.

If *inMethodName* is not fully qualified, only the currently imported libraries are checked for the given method name.

If *outLibName* is passed in, it receives the name of the library in which the method is found, or an empty string if the method is not found. If *inMethodName* is the name of an inline method, *outLibName* will be "global".

nil pointer

version 4.0

nil pointer → Pointer

Parameter	Type	Description
Result	Pointer	← A nil pointer

Discussion

This command returns a nil pointer (big surprise). It is designed to be used in those cases where you want to specifically pass a nil pointer to a method, instead of using the standard trick:

```
// old way
c_pointer($nil)
doSomething($nil)

// new way
doSomething(nil pointer)
```

With this command your intention is clearer.

redirect**version 1**
modified v5

```
redirect(inURL {; inIsPermanent})
```

Parameter	Type	Description
inURL	Text	→ Destination URL
inIsPermanent	Boolean	→ True to generate 301 status

Discussion

This command issues a true HTTP redirect to the specified URL, generating the necessary response status, headers and content based on the HTTP version of the client.

inURL may be a full URL to an external server (beginning with "http://"), an absolute path, or a relative path. Absolute paths are considered to be relative to the web root. Relative paths are relative to the file which contains the **redirect** command.

If *inIsPermanent* is passed and is *True*, a *301 Moved Permanently* status is returned to the browser instead of *303 See Other*.

As soon as **redirect** is executed, the script immediately stops execution and returns control to the shell. Thus **redirect** will be the last command within a flow of execution.

require**version 1**

```
require(inPath)
```

Parameter	Type	Description
inPath	Text	→ Path of file to include

Discussion

This command is identical to the **include** command, but if the given file has already been loaded with the **require** command within the current script execution, it will not be loaded again.

The primary use for this command is to create a file of defined constants which you want to reference throughout your pages. For an example of this usage, see "define" on page 242.

RESOLVE POINTER**version 1
modified v5**

RESOLVE POINTER(inPointer; outName; outTableNum; outFieldNum)

Parameter	Type	Description
inPointer	Pointer	→ Pointer to resolve
outName	Text	← Receives referent name
outTableNum	Number	← Receives table number/array index
outFieldNum	Number	← Receives field number

Discussion

This command performs the same function as the **RESOLVE POINTER** command in 4D. Unlike 4D however, if *inPointer* points to an interprocess variable, *outName* will always be prefixed with "<>", even on macOS.

sleep**version 1**

sleep(inTicks)

Parameter	Type	Description
inTicks	Number	→ Ticks to delay

Discussion

This command is equivalent to:

```
delay process(current process; $inTicks)
```

The difference is that it will actually delay, whereas the equivalent 4D code, when executed from a web process, will do nothing.

throw**version 3.0**
modified v6.0

throw(inMessage | inCode {; inMessage})

Parameter	Type	Description
inMessage	Text	→ Error message
inCode	Longint	→ Error code

Discussion

This command generates an Active4D runtime error with the given error code and message, which will trigger the normal error handling mechanism and execute an error page if one has been set.

The first parameter may be either a code or a message. If it is a code, then a message may be passed in the second parameter.

Within an error handler, the code and message passed to **throw** can be retrieved with **get throw code** and **get throw message**.

time to longint**version 1**

time to longint(inTime) → Longint

Parameter	Type	Description
inTime	Time	→ Value to convert to a Longint
Result	Longint	← Converted value

Discussion

This command converts a Time value into a Longint value.

Unless you use a compiler declaration, Active4D's variables are variant in their type. To assign a Time to a variant variable as a Longint you must use this command. This is specially useful when you are storing Times in a Longint array and you want to set the elements as Times.

Note: You could also use **C_LONGINT** to fix a variable's type to Longint and assign a Time to it. This would do type conversion for you.

Math

Active4D adds several utility commands that solve simple but common math problems with a minimum of code.

max of**version 1**

max of(inValue1; inValue2) → Number

Parameter	Type	Description
inValue1	Number	→ Number to compare
inValue2	Number	→ Number to compare
Result	Number	← Greater of the two values

Discussion

This command compares *inValue1* to *inValue2*. The greater of the two values is returned.

min of**version 1**

min of(inValue1; inValue2) → Number

Parameter	Type	Description
inValue1	Number	→ Number to compare
inValue2	Number	→ Number to compare
Result	Number	← Lesser of the two values

Discussion

This command compares *inValue1* to *inValue2*. The lesser of the two values is returned.

random between**version 1 (modified v2)**

random between(inMin; inMax) → Number

Parameter	Type	Description
inMin	Number	→ Minimum number to return
inMax	Number	→ Maximum number to return
Result	Number	← Random number between inMin and inMax inclusive

Discussion

This command returns a random number between *inMin* and *inMax* inclusive. If *inMin* > *inMax* the result is undefined. Both *inMin* and *inMax* may be numbers in the full range of a 4D Real.

ObjectTools

Active4D allows you to convert ObjectTools objects to Active4D collections and vice versa. In addition, you can clear an ObjectTools object within Active4D.

Note: ObjectTools support in Active4D requires ObjectTools v4.0 or later.

clear object

v6.0

```
clear object(inObject)
```

Parameter	Type	Description
inObject	Number	→ ObjectTools handle

Discussion

This command is the equivalent of the **OT Clear** command in ObjectTools. After using this command the object can no longer be used by ObjectTools.

collection to object

v6.0

```
collection to object(inCollection) → Longint
```

Parameter	Type	Description
inCollection	Longint	→ Collection to convert
Result	Longint	← ObjectTools object handle

Discussion

This command converts the collection *inCollection* to a new ObjectTools object and returns the handle to the object. You may then use the object handle with any ObjectTools command.

Embedded collections within *inCollection* (whether in longints or longint arrays) are recursively converted to embedded objects.

object to collection

v6.0

```
object to collection(inObject) → Longint
```

Parameter	Type	Description
inObject	Longint	→ Object to convert
Result	Longint	← Collection handle

Discussion

This command converts the object *inObject* to a new Active4D collection and returns the handle to the collection. Embedded objects within *inObject* are recursively converted to embedded collections within the collection.

Pictures

Active4D allows you to read, manipulate, convert and write pictures to disk and to the response buffer. Using this capability, you can programmatically create and serve graphics on the fly.

The following 4D picture commands are supported in Active4D:

Picture size
PICTURE PROPERTIES
READ PICTURE FILE
WRITE PICTURE FILE
GET PICTURE FROM LIBRARY

As with all document commands, **READ PICTURE FILE** and **WRITE PICTURE FILE** have their paths checked against the list of safe document directories.

In addition to these commands, you may also use the following operators on pictures:

+ *horizontal concatenation*
/ *vertical concatenation*
+= *horizontally concatenate and assign*
/= *vertically concatenate and assign*

Here's an example of using the picture operators:

```
get picture from library(2000; $pict)
$pict := $pict + $pict
get picture from library(2001; $pict)
$pict += $pict     // this is equivalent to the assignment above
```

Using the image.a4d Script

In addition to the image commands and operators, there is a standard script file called `image.a4d` which you can use to load images dynamically, as well as to generate thumbnails from images. For documentation on this script, see below.

image.a4d (script file)

version 4.0
modified v5

image.a4d

This is not actually a command, but rather a script file that acts like a method. It is designed to be “called” by making it the image source in an HTML tag. You pass parameters to the script through the query string.

This script can dynamically load images from four different sources:

- Picture library
- Database
- Disk file
- Active4D method call
- 4D method call

In addition, you may request that the image be turned into a thumbnail of a given size, and you may request the image in PNG, JPEG or GIF format. These two features are controlled by two optional query string parameters:

Parameter	Description
size	Desired pixel width/height
width	Desired pixel width
height	Desired pixel height
format	“png”, “jpg” or “gif”

If *size*, *width* or *height* is passed, the image will be scaled proportionally to the given size. If *format* is passed, the default format will be overridden and the image will be returned in the requested format.

Loading from the 4D picture library

To load an image from the 4D picture library, pass the following query parameter:

Parameter	Description
id	Numeric id or name of picture

If this query parameter is passed, the picture with the given numeric id (if the first character is a digit) or name will be returned if it exists. By default the picture will be converted to PNG.

Loading from the database via query

To load an image from a database field via a query, pass the following query parameters:

Parameter	Description
<code>img_field</code>	Full [table]field reference to image field
<code>qry_field</code>	Field name of field to query
<code>qry_value</code>	Value to query <code>qry_field</code> for

If these three query parameters are passed, a query is done on `qry_field` for `qry_value`, and if one or more records results, the value of `img_field` from the first record is used. The field referenced by `qry_field` must be either Alpha or Longint (it's supposed to be an id field). By default the picture will be converted to JPEG.

Loading from the database via record number

To load an image from a database field via a record number, pass the following query parameters:

Parameter	Description
<code>img_field</code>	Full [table]field reference to image field
<code>recnum</code>	Record number of record to use

If these two query parameters are passed, the record with the given number is loaded, and if such a record exists, the value of `img_field` is used. By default the picture will be converted to JPEG.

Loading from a file

To load an image from a disk file, pass the following query parameter:

Parameter	Description
<code>file</code>	Web root-relative URL path

If this query parameter is passed, the image at the given location will be returned. By default the image will take the format of the original, depending on the filename extension. If the filename extension is not ".png" or ".gif", it defaults to JPEG format.

This use of image.a4d is primarily useful if you want to create a thumbnail, because otherwise it would be much more efficient to reference the image directly.

Loading from an Active4D method call

To load an image via an Active4D method call, pass the following query parameters:

Parameter	Description
<code>method</code>	Name of an Active4D method to call
<code>param</code>	Optional parameters to pass to the method

If the “method” query parameter is passed, the Active4D method with the given name will be called. The method should have the following signature:

```
method "getImage"($inParams; &$ioFormat)
  // Get a picture into $pict. You can optionally
  // set the format by setting $ioFormat.
  return ($pict)
end method
```

Of course you may change “getImage” to whatever you like. If you need to pass multiple parameters to the method, you can pack them together with **concat** (do not use “;” as the separator, use a character such as “|”), and then unpack them with **slice string** or **split string** in the method.

Loading from a 4D method call

To load an image via a 4D method call, pass the following query parameters:

Parameter	Description
method4d	Name of a 4D method to call
param	Optional parameters to pass to the method

If the “method4d” query parameter is passed, the 4D method with the given name will be called. The method should have the following signature:

```
`getImage($inParams)
C_TEXT($1)
C_PICTURE($0)

`Get a picture somehow, assign to $0
```

Of course the name of the method may be whatever you like. If you need to pass multiple parameters to the method, you can pack them together with **concat** and then unpack them in the 4D method.

Examples

Get a picture from the picture library in JPEG format:

```

```

Get an image from a table via a query, making a thumbnail 48 pixels in size:

```
<%
  $qry := build query string("; "img_field"; "[images]image"; \\
                                "qry_field"; "id"; \\
                                "qry_value"; 100; \\
                                "size"; 48)
%>

```

Get an image from a table via a record number:

```
<%
  $qry := build query string("; "img_field"; "[images]image"; \\
                                "recnum"; 27)
%>

```

Get an image from a file, making a thumbnail 128 pixels in size:

```
<% $qry := build query string("; "file"; "img/myimage.jpg"; \\
                                "size"; 128)
%>

```

Get an image from a method call, passing the value 100:

```
<% $qry := build query string("; "method"; "img.getImage"; \\
                                "param"; 100)

```

write gif

version 2
modified v4.0

```
write gif(inPicture {; inWidth {; inHeight{}})
```

Parameter	Type	Description
inPicture	Picture	→ Picture to convert to GIF format
inWidth	Number	→ Pixel width or scaling percentage
inHeight	Number	→ Pixel height or scaling percentage

Discussion

This command does the following:

- Scales *inPicture* according to *inWidth/inHeight* if passed
- Converts *inPicture* into a GIF graphic

- Replaces the current contents of the response buffer with the graphic
- Sets the content type of the response to “image/gif”

If *inWidth* is nonzero and *inHeight* is zero or is omitted, the height will be scaled proportionally to *inWidth*. If *inWidth* is zero and *inHeight* is nonzero, the width will be scaled proportionally to *inHeight*.

If *inWidth* or *inHeight* is a positive number, it is treated as an absolute pixel size. If *inWidth* or *inHeight* or *negative*, they are treated as scaling factors, where .5=50%, 2=200%, etc.

Note: 4D’s GIF conversion routines are optimized for pictures with 256 colors or less. Pictures with more colors will be converted with noticeable banding and dithering artifacts.

write jpeg

**version 2
modified v5**

```
write jpeg(inPicture {; inWidth {; inHeight{}}
```

Parameter	Type	Description
inPicture	Picture	→ Picture to convert to JPEG format
inWidth	Number	→ Pixel width or scaling percentage
inHeight	Number	→ Pixel height or scaling percentage

Discussion

This command does the following:

- Scales *inPicture* according to *inWidth/inHeight* if passed
- Converts *inPicture* into a JPEG graphic
- Replace the current contents of the response buffer with the graphic
- Set the content type of the response to “image/jpeg”

If *inWidth* is nonzero and *inHeight* is zero or is omitted, the height will be scaled proportionally to *inWidth*. If *inWidth* is zero and *inHeight* is nonzero, the width will be scaled proportionally to *inHeight*.

If *inWidth* or *inHeight* is a positive number, it is treated as an absolute pixel size. If *inWidth* or *inHeight* or *negative*, they are treated as scaling factors, where .5=50%, 2=200%, etc.

Note: As of v5, 4D’s built in picture conversion is being used to convert to JPEG, and the result is lower quality than what Active4D v4.x produced. PNG is recommended as the picture format for images that are not photographs.

write jpg**version 2
modified v5**

```
write jpg(inPicture {; inWidth {; inHeight{}})
```

Parameter	Type	Description
inPicture	Picture	→ Picture to convert to JPEG format
inWidth	Number	→ Pixel width or scaling percentage
inHeight	Number	→ Pixel height or scaling percentage

Discussion

This command is a synonym for **write jpeg**.

write png**version 4.0
modified v5**

```
write png(inPicture {; inWidth {; inHeight{}})
```

Parameter	Type	Description
inPicture	Picture	→ Picture to convert to PNG format
inWidth	Number	→ Pixel width or scaling percentage
inHeight	Number	→ Pixel height or scaling percentage

Discussion

This command does the following:

- Scales *inPicture* according to *inWidth/inHeight* if passed
- Converts *inPicture* into a PNG graphic
- Replace the current contents of the response buffer with the graphic
- Set the content type of the response to "image/png"

If the original image within *inPicture* was a PNG image with an alpha channel, the alpha channel is preserved in the output of this command.

If *inWidth* is nonzero and *inHeight* is zero or is omitted, the height will be scaled proportionally to *inWidth*. If *inWidth* is zero and *inHeight* is nonzero, the width will be scaled proportionally to *inHeight*.

If *inWidth* or *inHeight* is a positive number, it is treated as an absolute pixel size. If *inWidth* or *inHeight* are negative, they are treated as scaling factors, where .5=50%, 2=200%, etc.

Queries

Active4D implements the extended (but undocumented) syntax of the **QUERY**, **QUERY SELECTION** and **ORDER BY** commands.

QUERY/QUERY SELECTION

version 2

QUERY/QUERY SELECTION(inTable; inConjunction; inField; inComparator; inValue {;*})

Parameter	Type	Description
inTable	Table	→ Table on which to query
inConjunction	Literal string	→ " ", "&" or "#"
inField	Field	→ Field on which to query
inComparator	Literal string	→ "=", "#", "<", "<=", ">" or ">="
inValue	<any>	→ Value to compare against field
*		→ To defer query

Discussion

The extended syntax of these commands puts the field, comparator and value into separate parameters. To dynamically build a query statement, you can build the statement as a string and then pass the string to the **EXECUTE** command.

ORDER BY

version 2

ORDER BY

Discussion

Active4D supports “built” sorts using multiple **ORDER BY** statements, in the same way that a built query can be executed. To build a sort, add a * parameter at the end of the statement. This will defer the sort until an **ORDER BY** statement is executed without a final * parameter.

As with built queries, once a deferred **ORDER BY** statement has been executed, the sort may be executed with the following syntax:

```
order by([MyTable])
```

ORDER BY FORMULA

version 2
modified v6.0r8

```
ORDER BY FORMULA({*; inTable {; inExpression {; > or <}}; inExpression2 ; > or <2 ; ... ;  
inExpressionN ; > or <N}
```

Discussion

If the initial * parameter is not passed, **ORDER BY FORMULA** executes within 4D's context:

- It executes on the Server, even if initiated from a Remote.
- You only have access to 4D's execution context — process/interprocess variables, 4D methods, etc. You do *not* have access to Active4D's context — local variables, built in collections, libraries, etc.

If the initial * parameter is passed, **ORDER BY FORMULA** executes within Active4D's context:

- It executes on the Server if initiated from Server, on Remote if initiated from a Remote. On Remote, this means that potentially every record in the selection will have to be loaded from the Server.
- You have full access to Active4D's context — local variables, built in collections, libraries, etc.

Examples

```
// Execute in 4D's context  
order by formula([contacts]; length([contacts]last_name))  
  
// Execute in Active4D's context so we can access the hashName  
// method in our textUtils library.  
order by formula(*; [contacts]; \  
textUtils.nameHash([contacts]last_name; [contacts]first_name))
```

Query Params

When a URL is requested which has a query string, Active4D puts the query parameter names and their associated values into a collection. You can access this collection in your scripts.

Note: If the “parameter mode” configuration option is set to “form variables”, query parameters will not go into the query params collection, but into the form variables collection. Likewise, if the “parameter mode” option is set to “query params”, form variables will go into the query params collection.

The query params collection is read-only, and follows the same pattern as all read-only collections.

Query Params Items

Normally, item values in the query params collection are either text or text arrays. However, if a form is posted which contains raw data (such as XML) and no form variables, and the “parameter mode” configuration option is set to “query params”, Active4D creates a single BLOB item in the query params collection with the key “_data_”.

Duplicate Query Parameters

If several query parameters have the same name, Active4D creates a text array to hold the values of the duplicate items, the key of which is the query parameter name.

Also, if the “parameter mode” configuration option is set to “query params”, multiple-choice form lists will end up in the query params collection. For more information on the ramifications of this, see “Multiple-choice Form Fields” on page 201.

_query**version 3.0**`_query` → Longint

Parameter	Type	Description
Result	Longint	← Iterator reference

Discussion

This command is an alias for the **query params** command.

query params**version 2**`query params` → Longint

Parameter	Type	Description
Result	Longint	← Iterator reference

Discussion

This command returns an iterator to the first item in the query params collection.

For more information on iterators, see “Iterators” on page 214.

query params has**version 2**`query params has(inKey {; *})` → Boolean

Parameter	Type	Description
inKey	Text	→ Key of item to test
*	*	→ Perform wildcard search
Result	Boolean	← True if key is in collection

Discussion

This command searches the query params collection for the item with the key *inKey*. If * is passed, *inKey* may contain 4D wildcard characters and they will be honored in the search.

If the item is found, *True* is returned, otherwise *False*.

get query param

version 2
modified v5

get query param(inKey {; inIndex}) → Text | BLOB

Parameter	Type	Description
inKey	Text	→ Key of query param to retrieve
inIndex	Longint	→ Index of multiple-choice list value to retrieve
Result	Text BLOB	← Value of query param or ""

Discussion

This command searches the query param collection for the item with the key *inKey*.

If the item is found and an index is not specified, the item's value is returned. If the item value is an array of multiple choices, the first multiple-choice value is returned. Note that this behavior is different than using the syntax:

```
$value := _query{"multiple_choice_value"} // Returns longint
```

When using the collection indexing syntax, a reference to an array with no index returns the array itself, which when assigned results in the current element of the array, which is a longint.

If the item is found and an index is specified, the given multiple choice value is returned. If the index is out of range, an error is generated and execution is aborted.

If the item is not found, an empty string is returned.

Note: The query param value is URL decoded and converted from UTF-8, so you do not need to perform any decoding.

get query param choices

version 2

get query param choices(inKey; outArray)

Parameter	Type	Description
inKey	Text	→ Key of query param to retrieve
outArray	String/Text Array	← Receives the array of choices

Discussion

This command searches the query params collection for the item with the key *inKey*.

If *outArray* is defined but is not a string or text array, an error is generated and execution is aborted. If *outArray* was not defined, it is created as a text array.

If the item is found and has multiple-choice values in an array, *outArray* is set to a copy of the multiple-choice values.

If the item is found and its value is not an array of multiple-choice values, *outArray* is set to a single element containing the query param value.

If the item is not found, *outArray* is resized to zero elements.

get query param count

version 2

get query param count(inKey) → Longint

Parameter	Type	Description
inKey	Text	→ Key of query param to check
Result	Longint	← Size of array

Discussion

You use this command to check how many values were selected in a multiple-choice list on a form.

This command searches the query params collection for the item with the key *inKey*.

If the item is found and has multiple-choice values in an array, the size of the array is returned.

If the item is found and its value is not an array of multiple-choice values, 1 is returned.

If the item is not found, zero is returned.

get query params

version 2
modified v5

```
get query params(outKeys {; outValues})
get query params{(*; inKeyFilter)} → Text
```

Parameter	Type	Description
outKeys	String/Text Array	← Receives the collection keys
outValues	String/Text Array	← Receives the collection values
*		→ Indicates a filter is being used
inKeyFilter	Text	→ Keys are matched against this
Result	Text	← Concatenation of keys and values

Discussion

This command has two forms. The first form fills *outKeys* and *outValues* with all of the keys and values in the query params collection.

If *outKeys* was not defined, it is created as a string array. If *outValues* was not defined, it is created as a text array.

If *outKeys* is defined but is not a string or text array, an error is generated and execution is aborted.

The second form of the command returns a concatenation of the form variables in the form "key1=value1;key2=value2", suitable for use as a query string. If a form variable is a multiple-choice array, all of the array values are included in the concatenation.

The keys and values are converted to UTF-8 and URL encoded.

You may optionally pass in a string which will be matched against items whose key matches the string. Wildcards are allowed in the filter string. To *include* items that match the filter, prefix it with '+'. To *exclude* items that match the filter, prefix it with '-'. If there is no prefix, it is assumed to be an inclusion filter.

Note: The keys and values are URL decoded and converted from UTF-8 to Unicode, so you do not need to perform any decoding.

count query params

version 2

count query params → Longint

Parameter	Type	Description
Result	Longint	← Number of items in collection

Discussion

This command returns the number of items in the query params collection.

build query string

version 3.0

build query string({*; inQuery; inName; inValue {...; inNameN; inValueN}) → Text

Parameter	Type	Description
*	*	→ If passed, don't suppress empty values
inQuery	Text	→ Existing query string to build on
inName	Text	→ Query param name
inValue	<any>	→ Query param value
Result	Text	← New query string

Discussion

This command is extremely useful for building a query string to add to a URL. It adds the given name/value pairs to *inQuery*, automatically converting the keys and values to UTF-8 and then URL encoding.

As of HTML 4.0, the recommended practice for delimiting query parameters is to use a semicolon, not an ampersand (&). Accordingly this is what **build query string** does.

If *inQuery* is empty, the result will begin with "?". If *inQuery* starts with "-", it can be followed by one or more of the following switches:

- **"e" - External reference mode:** Use this mode if you are passing a query string to a non-Active4D server that may not understand semicolons as query parameter delimiters. If this switch is set, by default query parameters will be delimited with "&"; as required by HTML 4+ for URLs embedded in HTML. If you are building a query string for use in an external redirect, be sure to set the "r" switch as well.
- **"r" - Redirect mode:** This switch has no effect unless the "e" switch is set as well, in which case the query parameter separator will be "&" instead of "&";.
- **"u" - URL encoded mode:** It is assumed the query parameter names and values are already URL encoded, so no encoding is done.
- **"p" - No-prefix mode:** No leading "?" or query parameter delimiter will be added. This switch overrides the "a" switch.

- **“a” - Append mode:** Whether or not *inQuery* is empty (not including the switches), it will be considered non-empty. Use this switch to append the results of this command to an existing query string.

Note: You do not need to use the “a” switch when creating a query to pass to `fusebox.makeURL` or `fusebox.handleError`.

If text follows the switches, there should be another “-” between the switches and the text.

If there are no switches set and *inQuery* is not empty, the name/value pairs will be appended to *inQuery*.

If the * parameter is not passed, name/value pairs with an empty value will be skipped. If the * parameter is passed, all name/value pairs will be appended to the query string.

Examples

```
$query := build query string(""); \\  
          "rec"; record number([employees]); \\  
          "action"; _form{"f_action"})  
// $query = "?rec=123;action=edit"  
redirect("edit_emp.a4d" + $query)  
  
// using the e and r switches  
  
$query := build query string("-e"; "foo"; 7; "bar"; 13)  
// $query = "?foo=7&bar=13"  
  
$query := build query string("-er"; "foo"; 7; "bar"; 13)  
// $query = "?foo=7&bar=13"  
  
$barValue := url encode query("this is a test")  
$query := build query string("-u"; "foo"; 7; "bar"; $barValue)  
// $query = "?foo=7;bar=this+is+a+test"  
  
$query := build query string("-p"; "foo"; 7; "bar"; 13)  
// $query = "foo=7;bar=13"  
redirect("foobar.a4d?" + $query)  
  
$query := build query string("-a"; "foo"; 7; "bar"; 13)  
// $query = ";foo=7;bar=13"  
redirect(fusebox.makeURL("foobar.main"; $query))
```

Regular Expressions

Active4D implements a powerful suite of regular expression commands that allow you to perform complex searches and manipulations on text.

Regular expressions are in fact a compact programming language, and as such allow you to perform text manipulations that would take many lines of code to implement.

Pattern Syntax

Regular expressions follow the ICU syntax, which is described here:

<http://userguide.icu-project.org/strings/regexp>

Regular expressions in Active4D fully support Unicode.

Note: Versions of Active4D previous to v5 used the PCRE library for regular expression matching. Although the pattern syntax of PCRE and ICU are very similar, there are some differences, so you should carefully check your regular expression patterns if you are upgrading from v3/v4.x to v6.

Regular expression patterns must be enclosed in delimiters, for example a forward slash (/). Any non-alphanumeric Unicode character in the Basic Multilingual Plane (other than backslash) can be used as the delimiter. If the delimiter character is used in the expression itself, it needs to be escaped by a backslash.

The ending delimiter may be followed by various modifiers that affect the matching. The pattern modifiers are discussed in detail here (see “Flag Options”):

<http://userguide.icu-project.org/strings/regexp#TOC-Flag-Options>

For examples of the pattern syntax, see the example code for the commands in this chapter.

Using Regular Expressions

Entire books can be (and have been) written about regular expressions. It is not within the scope of this document to give any kind of tutorial on their usage. A web search will turn up lots of resources for learning all of the amazing uses of regular expressions.

regex callback replace

version 3.0

```
regex callback replace(inPattern; inSubject; inCallback; outResults {; inLimit})
```

Parameter	Type	Description
inPattern	Text String/Text Array	→ Search pattern(s)
inSubject	Text/BLOB String/Text Array	→ Subject(s) to search
inCallback	Text	→ Name of method to call
outResults	Text String/Text Array	→ Receives the replaced text
inLimit	Number	→ Limit on number of matches

Discussion

This command is almost identical to **regex replace**, except that instead of replacement parameter, you specify the name of a callback method that will be called once for each match.

The method must be an Active4D method which takes a single reference array parameter. The array will contain the matched elements in the subject string, with the entire matched string in element zero and captured subpatterns in subsequent elements. The callback should return the entire replacement string.

Note: Internally this command creates a local array called `$_a4d_regex_callback_array__`, which is passed to the callback method. Be sure not to give any of your local variables this name.

Example

Let's say you want to add one year to a bunch of dates.

```
method "addOneYear"(&$inArray)
  c_longint($year; $month; $day)
  $month := num($inArray{1})
  $day := num($inArray{2})
  $year := num($inArray{3})
  $date := add to date(!00/00/00!; $year + 1; $month; $day)
  return (string($date; MM DD YYYY Forced))
end method

/*
The pattern looks for the digit 0 or 1,
followed by any digit,
followed by a forward slash,
followed by the digits 0-3,
followed by any digit,
followed by a forward slash,
followed by four digits.
*/
$pattern := "|([01]\d)/([0-3]\d)/(\d{4})|"
array text($dates; 0)
set array($dates; "08/27/2003"; "03/30/2003")
regex callback replace($pattern; $dates; "AddOneYear"; $results)
writebr(join array($results; "<br />\n"))

// Output is:
08/27/2004
03/30/2004
```

regex find all in array**version 3.0**

regex find all in array(*inArray*; *inPattern*; *outIndexes* {; *inStartIndex*) → Longint

Parameter	Type	Description
<i>inArray</i>	String/Text Array	→ Array to search
<i>inPattern</i>	Text	→ What to match in the array
<i>outIndexes</i>	Longint Array	← Receives the match indexes
<i>inStartIndex</i>	Longint	→ Where to start searching
Result	Longint	← Index of first matching element

Discussion

This command searches *inArray* for the all elements that match the pattern *inPattern*. The index of the first matching element is returned, or -1 if no elements match the pattern. The indexes of all matching elements are put in the array *outIndexes*.

outMatches does not have to be defined before using this command.

Example

```
// Find all elements that start with "foo" or end with "bar"
array text($array; 0)
set array($array; "one foo"; "two bar"; "fool"; "bart")
$index := regex find all in array($array; "/^foo|bar$/"; \\
                                $matches)
writebr(join array($matches; "<br />\n"))

// Output is:
2
3
```

regex find in array**version 3.0**

regex find in array(*inArray*; *inPattern* {; *inStartIndex*}) → Longint

Parameter	Type	Description
<i>inArray</i>	String/Text Array	→ Array to search
<i>inPattern</i>	Text	→ What to match in the array
<i>inStartIndex</i>	Longint	→ Where to start searching
Result	Longint	← Index of first matching element

Discussion

This

command searches *inArray* for the first element that matches the pattern *inPattern*. The index of the first matching element is returned, or -1 if no elements match the pattern.

Example

```
// Find the first element that starts with "foo"
// or ends with "bar"
array text($array; 0)
set array($array; "one foo"; "two bar"; "fool"; "bart")
$index := regex find in array($array; "/^foo|bar$/")
writebr($index)

// Output is:
2
```

regex match

version 3.0
modified v5

regex match(inPattern; inSubject {; outMatches}) → Boolean

Parameter	Type	Description
inPattern	Text	→ What to match in the subject
inSubject	Text/BLOB	→ The text to search
outMatches	String/Text Array	← Receives the matches
Result	Boolean	← True if any matches were found

Discussion

This command searches *inSubject* for the *first* match to the regular expression given in *inPattern*. If a matches are found, *True* is returned, otherwise *False*.

If *inSubject* is a BLOB, it is assumed to be in the format *UTF8 Text without length*.

If *outMatches* is provided, it is filled with the results of the search. *\$outMatches{0}* will contain the text that matched the full pattern, *\$outMatches{1}* will contain the text that matched the first captured parenthesized subpattern, and so on.

If *outMatches* is a local variable (or collection item) and was not defined, it is created as a text array.

Example

Splitting a URL into constituent parts:

```
/*
  The following pattern will split a URL into six parts:
  1) "http://" if present
  2) hostname
  3) "/4dcgi" if present
  4) resource path
  5) "?" at start of query string if present
  6) query string if present
*/
$pattern := "|^(http://)?(?:[^\s/]+)(/4dcgi)?(?:[^\s/]+)(\?)?(.*)|i"
$url := "http://www.myserver.com/4dcgi/index.a4d?foo=bar"
$found := regex match($pattern; $url; $matches)
writebr(join array($matches; "<br />\n"; 0))

// Output is:
http://www.myserver.com/4dcgi/index.a4d?foo=bar
http://
www.myserver.com
/4dcgi
/index.a4d
?
foo=bar
```

Notice in the above example that `$matches{0}` contains the full URL, because that is what matched the full pattern.

regex match all

version 3.0

regex match all(inPattern; inSubject; outMatches) → Longint

Parameter	Type	Description
inPattern	Text	→ What to match in the subject
inSubject	Text/BLOB	→ The text to search
outMatches	Collection	← Receives the matches
Result	Longint	← The number of matches made

Discussion

This command searches *inSubject* for the *all* matches to the regular expression given in *inPattern*. After the first match, subsequent matches are made starting from the end of the previous match. The number of matches made is returned.

If *inSubject* is a BLOB, it is assumed to be in the format *UTF8 Text without length*.

outMatches is filled with the results of the search. The collection will contain one item for each match, with the keys named from "00001" to the number of matches made.

Note: Obviously because of the naming scheme you are limited to 99,999 matches.

The contents of each collection item is an array which contains what a call to **regex match** would return, i.e. `$outMatches{$key}{0}` will contain the text that matched the full pattern, `$outMatches{$key}{1}` will contain the text that matched the first captured parenthesized subpattern, and so on.

Example

Find matching HTML tags:

```
$pattern := "|(<([\w]+)[^>]*>)(.*)(</\2>)|"
$html := "<b>example: </b><p>this is a test</p>"
regex match all($pattern; $html; $matches)

for each($matches; $key)
    $html := join array($matches{$key}; ", "; 0; false; true)
    writebr($html; ""; A4D Encoding All)
end for each

// Output is:
"<b>example: </b>", "<b>", "b", "example: ", "</b>"
"<p>this is a test</p>", "<p>", "p", "this is a test", "</p>"
```

regex quote pattern

version 3.0

`regex quote pattern(inPattern {; inDelimiter}) → Text`

Parameter	Type	Description
<code>inString</code>	Text	→ Text to quote
<code>inDelimiter</code>	Char	→ Additional character to quote
<code>Result</code>	Text	← Quoted text

Discussion

This command puts a backslash in front of every character in *inString* that is part of the regular expression syntax. This is useful if you have a dynamically generated string that you need to match in some text and the string may contain special regex characters.

If *inDelimiter* is specified, it will also be escaped. This is useful for escaping the delimiter that is used in your regex patterns, such as `"/`.

The special regular expression characters are:

```
. \ + * ? [ ^ ] $ ( ) { } = ! < > | : .
```

Example

Let's say you are going to do a regex match on a string entered by the user in the form field `"f_find"`. You would need to do something like this:

```
selection to array([Contacts]Last; $lastNames)
$pattern := "/" + regex quote pattern(_form{"f_find"}; "/") + "/" + "/i"
$index := regex find in array($lastNames; $pattern)
```

regex replace

version 3.0
modified v5

regex replace(inPattern; inSubject; inReplacement; outResult {; inLimit})

Parameter	Type	Description
inPattern	Text String/Text Array	→ Search pattern(s)
inSubject	Text/BLOB String/Text Array	→ Source text to search
inReplacement	Text String/Text Array	→ Replacement text
outResult	Text/BLOB String/Text Array	→ Receives replaced text
inLimit	Number	→ Limit on number of matches

Discussion

This command searches *inSubject* for matches to *inPattern* and replaces them with *inReplacement*. If *inLimit* is specified, only *inLimit* matches will be replaced. If *inLimit* is omitted or is ≤ 0 , all matches are replaced.

If matches are found, the new subject(s) will be returned in *outResult*, otherwise the subject(s) will be returned unchanged. *inSubject* and *outResult* may be the same variable.

If *inSubject* is a BLOB, then *outResult* must also be a BLOB, and they are both assumed to be in the format *UTF8 Text without length*. Note that *outResult* may not be an element of a BLOB array.

Every parameter to **regex replace** (except *inLimit*) can be an array.

If *inSubject* is an array, the search and replace is performed on every element of *inSubject*, and *outResult* will be an array with the same number of elements.

If *inPattern* and *inReplacement* are arrays, then **regex replace** walks through the arrays in parallel and uses the corresponding elements to do a search and replace on *inSubject*. If *inReplacement* has fewer elements than *inPattern*, an empty string is used for the remaining replacement values.

If *inPattern* is an array and *inReplacement* is a string, **regex replace** searches *inSubject* for each pattern and replaces with the replacement string.

If *inPattern* is a string and *inReplacement* is an array, it is an error, as this does not make sense.

Replacement Syntax

The real power in **regex replace** lies in the replacement syntax. In addition to referencing matched strings, you may also manipulate the case of the result.

In the case of the $\backslash W$ notation, $\backslash 0$ returns the entire matched pattern, $\backslash 1$ returns the first captured subpattern, and so on. Thus $\backslash 0$ and $\&$ are equivalent.

Notation	Action
<code>\N</code>	Returns the Nth captured subpattern
<code>\{N}</code>	Returns the Nth captured subpattern
<code>\<pattern></code>	Returns the named subpattern
<code>&</code>	Returns the entire matched pattern
<code>\U</code>	Starts an uppercase run
<code>\L</code>	Starts a lowercase run
<code>\E</code>	Ends an uppercase or lowercase run
<code>\u</code>	Uppercases the next letter
<code>\l (lowercase L)</code>	Lowercases the next letter

If you are using the `\N` notation and it is directly followed by a number, you must use the `\{N}` form to separate the pattern number from the number that follows. For example, if you want to replace a match with the first captured subpattern followed by the number 1, you must do it this way:

```
\{1}1
```

If your replacement pattern contains the character `'&'` and you do not intend it to be replaced with the entire matched pattern, it must be preceded with a backslash.

The `\U` and `\L` markers begin a case-changing run. Every character between these markers and either a `\E` marker or the end of the replacement has its case changed accordingly. Pattern and subpattern substitutions are done *before* case changes so you can change the case of matched patterns.

For example, the replacement text:

```
\U&\E \L\1\E
```

would uppercase the entire matched pattern and lowercase the first captured subpattern.

The `\u` and `\l` markers are similar, but they change the case of the next character only. So the replacement syntax:

```
\u& \l\1
```

would uppercase the first character of the entire matched pattern and lowercase the first character of the first matched subpattern.

The `/e` Pattern Option

If the search pattern included the `/e` option, after all replacements are done, the replacement text is executed as Active4D code and the result is used as the replacement text.

For example, this replacement text would return the first two characters of the entire matched pattern:

```
return (substring("\U&\E"; 1; 2))
```

If the entire matched pattern is "John Doe", this expression would first resolve to:

```
return (substring("JOHN DOE"; 1; 2))
```

Then this text would be executed, which would return "JO" as the replacement text.

Note a few important things in this example:

- You must use **return** to return the value to **regex replace**
- When using a match pattern in an expression, be sure to enclose it in double quotes.
- When you specify the replacement text, it is usually enclosed in double quotes, so you must escape double quotes within the replacement text like this:

```
$replace := "return (substring(\"\U&\E\"; 1; 2))"
```

Because the replacement text is executed as Active4D code within the context of the current execution, you may do anything you wish, including:

- Use any supported commands.
- Call Active4D and 4D methods.
- Access any local variables that were available within the scope of the **regex replace** command.
- Access any of the built-in collections such as form variables and query params.
- Execute more than one line of code by separating the lines by "\r", as long as the last line executes a **return** statement.

Examples

This example uses the `/e` pattern modifier to lowercase HTML tags in the subject:

```
$pattern := "/(<\/?)(\w+)([^\>]*>)/e"
$replace := "return (\\"1\" + lowercase(\"2\") + \"3\")"
regex replace($pattern; "<EM>test</EM>"; $replace; $result)
writebr($result; " "; A4D Encoding All)

// Output is:
<em>test</em>
```

This example replaces all occurrences of “foo” at the beginning of the subject with “bar”, and replaces all occurrences of “bar” at the end of the subject with “foobar”:

```
array text($patterns; 0)
set array($patterns; "/^foo(\w*)/"; "/(\w*)bar$/")
array text($subjects; 0)
set array($subjects; "foo is bar"; "fool's gold"; "\
    raise the bar"; "rebar")
array text($replacements; 0)
set array($replacements; "bar"; "foobar")
regex replace($patterns; $subjects; $replacements; $results)
writebr(join array($results; "<br />\n"))

// Output is:
bar is foobar
bar's gold
raise the foobar
foobar

// using named subpatterns
$pattern := "/(?P<area>\d{3})-?(?P<exch>\d{3})-?(?P<num>\d{4})/"
$replace := "(\<area>) \<exch>- \<num>"
regex replace($pattern; "5551234567"; $replace; $result)

// $result = "(555) 123-4567"
```

regex split

version 3.0

regex split(inPattern; inSubject; outResults {; inLimit {; inFlags}) → Number

Parameter	Type	Description
inPattern	Text	→ Delimiter pattern
inSubject	Text/BLOB	→ Source text to split
outResults	String/Text Array	→ Receives split strings
inLimit	Number	→ Limit on number of matches
inFlags	Number	→ Controls behavior of split
Result	Number	← Number of split strings

Discussion

This command is similar to the **split string** command, but *inSubject* is split along the boundaries matched by *inPattern*.

If *inLimit* is specified, then only substrings up to *inLimit* are returned, and if *inLimit* is ≤ 0 , it actually means “no limit”, which is useful when you want to pass *inFlags* as well.

inFlags can be any combination of the following flags, which may be combined with bitwise | operator:

Flag	Effect
A4D Regex Split No Empty	Only non-empty pieces will be returned
A4D Regex Split Capture Delims	Parenthesized expressions in the delimiter pattern will be captured and returned

Example

This example will split a string into its characters:

```
$str := "string"
regex split("//"; $str; $chars; 0; A4D Regex Split No Empty)
writebr(join array($chars; "/"))

// Output is:
s/t/r/i/n/g
```

Request Cookies

Active4D puts client cookies and their associated values into a collection. You can access this collection in your scripts.

The keys in the request cookies collection are text. The official specification for cookie keys says that they must be all ASCII with no whitespace; however, in practice this may not be the case, so Active4D performs the following transforms on cookie keys to make them compliant:

- The text is URL decoded to UTF-8. For example, "%20" becomes a space.
- Diacriticals are removed. For example, "côte" becomes "cote".
- Non-ASCII Latin characters are transformed to ASCII equivalents. For example, "æ" becomes "ae".
- The text is converted to ASCII. All non-ASCII characters are converted to "?".
- Whitespace is replaced with "_".

For example, the cookie key "côte ægis¥" is transformed to "cote_aegis?". Note that the transforms are only done when the cookie is received; the sender's cookie remains unaffected.

The values in the request cookies collection are all text. Note that no character set conversions are done on this text other than URL decoding to UTF-8; it is considered opaque by Active4D. It is up to you to properly encode cookie values when you set them.

The request cookies collection is read-only, and follows the same pattern as all read-only collections.

request cookies

version 2

request cookies → Longint

Parameter	Type	Description
Result	Longint	← Iterator reference

Discussion

This command returns an iterator to the first item in the request cookies collection.

For more information on iterators, see “Iterators” on page 214.

get request cookie

**version 2
modified v5**

get request cookie(inKey) → Text

Parameter	Type	Description
inKey	Text	→ Key of request cookie to retrieve
Result	Text	← Value of request cookie or ""

Discussion

This command searches the request cookie collection for the item with the key *inKey*.

If the item is found, the item’s value is returned.

If the item is not found, an empty string is returned.

Note: The cookie value is URL decoded and converted from UTF-8 to Unicode, so you do not need to perform any decoding.

get request cookies

version 2
modified v5

get request cookies(outKeys {; outValues})

Parameter	Type	Description
outKeys	String/Text Array	← Receives the collection keys
outValues	String/Text Array	← Receives the collection values

Discussion

This command fills *outKeys* and *outValues* with all of the keys and values in the request cookies collection.

If *outKeys* was not defined, it is created as a string array. If *outValues* was not defined, it is created as a text array.

If *outKeys* is defined but is not a string or text array, an error is generated and execution is aborted.

Note: The cookie keys and values are URL decoded and converted from UTF-8 to Unicode, so you do not need to perform any decoding.

count request cookies

version 2

count request cookies → Longint

Parameter	Type	Description
Result	Longint	← Number of items in collection

Discussion

This command returns the number of items in the request infos collection.

Request Info

Every time Active4D receives an HTTP request, a number of headers are passed as part of the request, such as the content length and user agent identifier.

Some of this information is commonly used, such as cookies, and is placed in specialized collections for easy access. *All* of the headers are put in the request info collection, which is accessible from your scripts. For more information on HTTP headers, see RFC 2616 at <http://www.w3.org/Protocols/>.

Request Info Collection Items

In addition to all of the HTTP headers, all of the information put into the request info array (see “A4D Execute <type> request” on page 66) is also put into the request info collection. The request info array elements appear in the request info collection as the following items:

Key	Value
*ajax	“1” if an XMLHttpRequest is made, “0” otherwise
*doctype	“doctype” option from Active4D.ini, either “html” or “xhtml”
*host	Hostname used in request without port
*host address	As set in 4D
*host port	As set in 4D, usually “80” for HTTP or “443” for SSL
*http version	“1.0” or “1.1”
*remote address	As set in 4D
*request method	The method used in the request (“GET”, “POST”, etc.)
*secure	As set in 4D, should be “1” for secure, “0” if not
*virtual host	Virtual host configured in VirtualHosts.ini

The request info collection is read-only, and follows the same pattern as all read-only collections. All of the values in the request info collection are text.

request info

version 2

request info → Longint

Parameter	Type	Description
Result	Longint	← Iterator reference

Discussion

This command returns an iterator to the first item in the request info collection.

For more information on iterators, see “Iterators” on page 214.

get request info

sversion 2

get request info(inKey) → Text

Parameter	Type	Description
inKey	Text	→ Key of request info to retrieve
Result	Text	← Value of request info or ""

Discussion

This command searches the request info collection for the item with the key *inKey*.

If the item is found, the item’s value is returned.

If the item is not found, an empty string is returned.

get request infos

version 2

get request infos(outKeys {; outValues})

Parameter	Type	Description
outKeys	String/Text Array	← Receives the collection keys
outValues	String/Text Array	← Receives the collection values

Discussion

This command fills *outKeys* and *outValues* with all of the keys and values in the request infos collection.

If *outKeys* was not defined, it is created as a string array. If *outValues* was not defined, it is created as a text array.

If *outKeys* is defined but is not a string or text array, an error is generated and execution is aborted.

count request infos

version 2

count request cookies → Longint

Parameter	Type	Description
Result	Longint	← Number of items in collection

Discussion

This command returns the number of items in the request cookies collection.

Request Value

Occasionally you may not know which collection a value is in or you don't care which collection it is in, you just know that it was part of the request. In such cases you can use this command.

get request value

version 2

get request value(inKey) → <any>

Parameter	Type	Description
inKey	Text	→ Key of item to retrieve
Result	<any>	← Value of item or ""

Discussion

This command searches these collections in order: query params, form variables, request cookies, request info.

If the item is found, the item's value is returned.

If the item is not found, an empty string is returned.

Resources

Active4D enhances the **Get indexed string** and **STRING LIST TO ARRAY** commands to allow you to work with other resource types (such as 4DK#) that are in STR# format..

Get indexed string

(modified 4D) version 6
modified v4.5

Get indexed string({*; inResType; } inResID; inIndex) → Text

Parameter	Type	Description
*	*	→ Indicates a resource type is passed
inResType	Text	→ Four-character resource type
inResID	Number	→ ID of resource
inIndex	Number	→ Index of string to get
Result	Text	← Requested string

Discussion

This command works just like the 4D version of the command, but it also allows you to retrieve strings from resources such as 4DK# which are in STR# format.

```
$s := get indexed string(*; "4DK*"; 10; 1)

// $s now contains something like "January:1:L".
// We want only the name.

$s := slice string($s; ":")
```

STRING LIST TO ARRAY**(modified 4D) version 6
modified v4.5**

STRING LIST TO ARRAY({*; inResType; } inResID; outStrings)

Parameter	Type	Description
*	*	→ Indicates a resource type is passed
inResType	Text	→ Four-character resource type
inResID	Number	→ ID of resource
outStrings	String/Text Array	← All strings in resource

Discussion

This command works just like the 4D version of the command, but it also allows you to retrieve strings from resources such as 4DK# which are in STR# format.

```
array string(15; $list; 0)
string list to array(*; "4DK*"; 10; $list)

// $s now contains strings like "January:1:L".
// We want only the name.

for ($i; 1; size of array($list))
  $list{$i} := slice string($list{$i}; ":")
end for
```

Response Buffer

When you want to return data to the client, you do so by appending text to the *response buffer*. This buffer becomes the body of the HTTP response.

The commands in this section allow you to write text and graphics to the response buffer, to get information about the size of the response buffer, and to control the character set conversion and encoding performed on text written to the response buffer.

buffer size

version 2
deprecated v5

buffer size → Longint

Parameter	Type	Description
Result	Longint	← Byte size of response buffer

Discussion

This command has been replaced by **response buffer size** and is no longer supported.

response buffer size

version 3.0

response buffer size → Longint

Parameter	Type	Description
Result	Longint	← Byte size of response buffer

Discussion

This command returns the current size of the response buffer in bytes.

clear buffer

version 2
deprecated v5

clear buffer

Discussion

This command has been replaced by **clear response buffer** and is no longer supported.

clear response buffer

version 3.0

clear response buffer

Discussion

This command completely clears the contents of the response buffer. Ordinarily you would have no need to use this command.

get response buffer

version 3.0
modified v5

get response buffer(outBuffer)

Parameter	Type	Description
outBuffer	Text or BLOB	← Contents of response buffer

Discussion

This command returns the current contents of the response buffer in *outBuffer*.

If **get response buffer** is used with a BLOB, the BLOB receives UTF-8 encoded text, as if you had executed **TEXT TO BLOB**(buffer; outBlob; *UTF8 Text without length*).

Note: Because execution is immediately terminated when binary data is written to the response buffer (e.g. with **write gif**), it is guaranteed that the result of this command will be text.

set response buffer

version 3.0
modified v5

set response buffer(inValue; inContentType)

Parameter	Type	Description
inValue	Text or BLOB	→ Value to write to the response buffer
inContentType	Text	→ Type of inValue's contents

Discussion

This command works the same as **write blob** but replaces the contents of the response buffer for text types instead of appending.

If **set response buffer** is used with a BLOB, the BLOB is assumed to contain text stored in the format *UTF8 Text without length*.

It is designed for use in post-processing the response in the *On Execute End* event handler. For example, you could use **get response buffer** to get the response, **regex replace** to process the buffer, and then **set response buffer** to use the processed buffer.

save output

version 3.0

save output(outBuffer)

Parameter	Type	Description
outBuffer	<variant>	→ Value that receives output

Discussion

This command sets *outBuffer* as the response buffer. All output that would normally end up going back to the browser (i.e. HTML and the output of the various **write** commands) will instead be appended to *outBuffer*. *outBuffer* must be either a variant value (such as a local variable or collection item) or an element of a string/text array.

Calls to **save output** must be balanced with a call to **end save output**, and the balancing call to **end save output** must be within the same scope as the corresponding call to **save output**.

You may nest calls to save output. This allows you to construct complex nested output, like this:

```
writebr("---> level 1")
save output($buffer)
  writebr("---> level 2")
  save output($buffer)
    writebr("---> level 3")
    writebr("<--- level 3")
  end save output
  write($buffer)
  writebr("<--- level 2")
end save output
write($buffer)
writebr("<--- level 1")
```

The output from the code above is:

```
---> level 1
---> level 2
---> level 3
<--- level 3
<--- level 2
<--- level 1
```

At each level, we save the output to *\$buffer*, then **write** it once we have restored the buffer.

Note: Output is not available in *outBuffer* until after **end save output** is executed.

end save output

version 3.0

end save output

Discussion

This command restores the output buffer that was current before the most recent call to **save output** and places the contents of the current output buffer into the target value specified in **save output**.

For more information, see “save output” on page 303.

set output charset

**version 2
modified v5**

set output charset(inCharset)

Parameter	Type	Description
inCharset	Number Text	→ Character set constant or name

Discussion

This command determines what character set conversion, if any, Active4D applies to text written to the response buffer in the current request. This conversion is applied after output character set encoding.

You may either pass one of the character set constants below or an IANA character set name. The output character set constants supported by Active4D are:

Character Set	Constant
UTF-8	A4D Charset UTF8
Macintosh Roman	A4D Charset None
Macintosh Roman	A4D Charset Mac
Windows Latin	A4D Charset Win
ISO-8859-1 (Latin1)	A4D Charset ISO Latin1

If you pass a name, it must be a valid IANA character set name. If the name is empty, it will default to “mac”. If it is invalid an error will be generated and execution will abort.

For more information on the output charset, see “Working with Character Sets” on page 104.

Note: For compatibility with previous versions, *A4D Charset None* is now effectively the same as *A4D Charset Mac*, because Unicode always has to be converted to a different encoding before being sent to the browser.

get output charset

version 2
modified v5

get output charset → Text

Parameter	Type	Description
Result	Text	← Character set name

Discussion

This command returns the name of the current output character set, which determines the character set to convert to when the response buffer is sent to the browser.

The name returned by this command is the internal, canonical name used by ICU, and thus may not be the same name you used either in Active4D.ini or with the **set output charset** command. For a complete list of charset names, see:

<http://demo.icu-project.org/icu-bin/convexp?s=IANA&s=ALL>

For more information on the output charset, see “Working with Character Sets” on page 104.

set output encoding

version 2

set output encoding(inEncoding)

Parameter	Type	Description
inEncoding	Number	→ How to encode special characters written to the response buffer

Discussion

This command determines what HTML character encoding, if any, Active4D applies to text written to the response buffer in the current request. This encoding is applied before output character set conversion.

The *inEncoding* parameter is a set of bit flags which specify the characters to encode. Each bit flag has a named constants defined. The constants (and their values) are:

- **A4D Encoding None (0):** No encoding is performed. You are responsible for manually encoding reserved characters, either by using the equivalent character entities directly or by using the **html encode** command.
- **A4D Encoding Quotes (1):** Single and double quotes are encoded.
- **A4D Encoding Tags (2):** The characters '`<`' and '`>`' are encoded.
- **A4D Encoding Ampersand (4):** The ampersand character ('`&`') is encoded.
- **A4D Encoding Extended (8):** All characters with an ASCII value ≥ 127 (non-breaking space and international characters) are encoded.
- **A4D Encoding HTML (8):** A synonym for *A4D Encoding Extended*. This is the default.
- **A4D Encoding All (65535):** All characters that have HTML character entities defined are encoded.

The default for Chinese and Japanese language systems is *A4D Encoding None*. The default for all other systems is *A4D Encoding HTML*, which encodes only non-breaking space and international characters. This allows you to use the `write` command to create HTML tags, while converting international characters.

For example, the following statement:

```
<% write("<td>Quelle bêtise!</td><td>&nbsp;</td>") %>
```

Would result in this output:

```
<td>Quelle b&ecirc;tise!</td><td>&nbsp;</td>
```

You can change the default output encoding with the "output encoding" option in `Active4D.ini`. For more information on output encoding, see "Output Encoding" on page 106.

get output encoding

version 2

get output encoding → Number

Parameter	Type	Description
Result	Number	← Set of bit flags

Discussion

This command returns the current set of bit flags which determine which characters to encode when writing to the response buffer. For more information on output encoding, see "Output Encoding" on page 106.

write**version 1**
modified v5

```
write(inValue {; inFormat {; inOutputEncoding}})
```

Parameter	Type	Description
inValue	<any>	→ Value to write to the response buffer
inFormat	Text Number	→ Format to use when converting to text
inOutputEncoding	Number	→ Character encoding of HTML special characters and

Discussion

This command works like the **String** command, but after converting *inValue* to text it appends the converted text to the response buffer. This is the primary method of generating dynamic HTML within Active4D code.

If you pass a Boolean value without a format, it will automatically output “True” or “False”, depending on the value passed in.

If a BLOB is passed to the **write** command, it is assumed to be text. You can specify the text format within the BLOB by passing the relevant constant (such as *UTF8 Text without length*) in the *inFormat* parameter. If no format is passed, the text format is assumed to be *UTF8 Text without length*.

If *inOutputEncoding* is passed, the text being written will be encoded according to the encoding style specified. This is a more convenient way of specifying a special encoding than bracketing a call to **write** with calls to **set output encoding**. You may pass * as a shortcut for *A4D Encoding All*. For more information on the value of this parameter, see “set output encoding” on page 305.

write blob**version 2**
modified version v5

```
write blob(inBlob; inContentType {; * | inCharset})
```

Parameter	Type	Description
inBlob	BLOB	→ BLOB to write to the response buffer
inContentType	Text	→ Type of inBlob’s contents
*	*	→ If passed, forces binary type
inCharset	Text	→ Character set of BLOB text

Discussion

This command does the following:

- Completely replaces the contents of the response buffer with the contents of *inBlob* if * is passed or if the content type does not begin with "text/".
- Sets the content type of the response based on *inContentType*, which may be either a MIME type or a file extension. If *inContentType* is a file extension (with or without leading dot), the corresponding MIME type is looked up from ExtensionMap.ini.
In either case, if the MIME type is not recognized, an error is generated and execution is aborted.
- Stops execution of the script if the MIME type is binary, i.e. if * is passed or if the type does not begin with "text/".
- If *inContentType* begins with "text/", it is assumed that the entire BLOB contains text stored in the format *Mac Text without length* or *UTF8 Text without length*.
- If *inContentType* begins with "text/" and * is not passed, *inCharset* can be an IANA character set name which indicates the charset of *inBlob*'s text. If *inCharset* is not passed, it defaults to UTF-8.

This command is designed to allow you to return binary data which you have created outside of Active4D. For example, you may create a JPEG thumbnail on the fly which you want to return as the response.

```
query([pictures];[pictures]name = $f_name)
$blob := MyCreateThumbnail([pictures]pict)
write blob($blob; "image/jpeg")
```

Previous to Active4D v5, **write blob** could also be used to write text greater than 32K in length. This is still possible in v6, but completely unnecessary, as string and text variables in v6 are capable of holding up to 2GB of text.

Note: It is up to you to ensure that *inContentType* matches the actual content type of *inBlob*.

writebr

version 1

```
writebr(inValue {; inFormat {; inOutputEncoding{}}
```

Parameter	Type	Description
inValue	<any>	→ Value to write to the response buffer
inFormat	Text Number	→ Format to use when converting to text
inOutputEncoding	Number	→ Character encoding of HTML special characters

Discussion

This command is a convenience routine for writing a value followed by an HTML line break and a line ending for the current platform (CR on Mac, CRLF on Windows). It is exactly equivalent to the following code:

```
write($inValue + "<br />" + $lineEnding; $inFormat)
```

writeln

version 1

```
writeln(inValue {; inFormat {; inOutputEncoding{}}
```

Parameter	Type	Description
inValue	<any>	→ Value to write to the response buffer
inFormat	Text Number	→ Format to use when converting to text
inOutputEncoding	Number	→ Character encoding of HTML special characters

Discussion

This command is a convenience routine for writing a value followed by a line ending for the current platform (CR on Mac, CRLF on Windows). It is exactly equivalent to the following code:

```
write($inValue + $lineEnding; $inFormat)
```

writep

version 1

```
writep(inValue {; inFormat {; inOutputEncoding{}}
```

Parameter	Type	Description
inValue	<any>	→ Value to write to the response buffer
inFormat	Text Number	→ Format to use when converting to text
inOutputEncoding	Number	→ Character encoding of HTML special characters

Discussion

This command is a convenience routine for writing a value followed by an HTML paragraph break and a line ending for the current platform (CR on Mac, CRLF on Windows). It is exactly equivalent to the following code:

```
write($inValue + "<p>" + $lineEnding; $inFormat)
```

Note: This command is not recommended, because it uses a form of HTML that is no longer compliant with current HTML standards.

write raw

version 2

```
write raw(inValue {; inFormat{}
```

Parameter	Type	Description
inValue	<any>	→ Value to write to the response buffer
inFormat	Text Number	→ Format to use when converting to text

Discussion

This command is equivalent to the **write** command, but it temporarily disables the current output encoding, thus the output encoding parameter is not available.

For example, if the current output encoding is *A4D Encoding All*, using the command

```
write("<b>This is bold</b>")
```

would not work as expected, because the output encoding would convert the HTML tags into HTML character entities, resulting in the following output: This would appear in

```
&lt;b&gt;This is bold&lt;/b&gt;
```

the browser as the literal string passed in, not as bold text as intended.

One could get the current output encoding, temporarily turn it off, then restore it, but that would quickly become cumbersome. By simply replacing **write** with **write raw** in the statement above, this problem can be quickly and easily avoided.

=

version 2
modified v6.0r4

```
=inExpression {; inFormat}
```

Parameter	Type	Description
inExpression	<any>	→ Expression to append to the response buffer
inFormat	<any>	→ Format to use with <i>inExpression</i>

Discussion

The = operator, followed by an expression, may be used at the beginning of the first line of an Active4D code block as a synonym for the **write** command. When used in this manner, the code block will exit after the first line of code executes.

Like the **write** command, the = operator will automatically convert the expression to text (using a format if you supply one); you do not have to explicitly use the **String** command where the **write** command would not require it.

This operator is very handy when using small Active4D code blocks embedded in a bunch of HTML. For example:

```
It is now <%= current time; hh mm am pm %> on <%=current date%>
```

Response Cookies

Active4D allows you to send *cookies* to the client browser. A cookie is basically a name and associated value which is stored on the client's machine. Cookies allow you to retain persistent information about a user across requests and sessions.

You send cookies to the client browser by adding them to the response cookies collection. This collection is read-write, and follows the same pattern as all read-write collections. All of the values in the response cookies collection are text and are considered opaque by Active4D. It is up to you to do any necessary URL encoding or character set conversion.

Cookie Fields

In addition to a name and a value, cookies can optionally have other attributes, including *domain*, *path*, and *expires*. The complete Netscape cookie specification can be found at:

<http://developer.netscape.com/docs/manuals/js/client/jsref/cookies.htm>

The only cookie attribute you will usually be concerned with is *expires*. If this attribute is not specified, a cookie expires when the browser is closed. To create a cookie that is retained across browser sessions, you must set the *expires* attribute to some date in the future. Active4D provides commands to do this.

response cookies

version 2

response cookies → Longint

Parameter	Type	Description
Result	Longint	← Iterator reference

Discussion

This command returns an iterator to the first item in the response cookies collection.

For more information on iterators, see “Iterators” on page 214.

get response cookie

**version 2
modified v5**

get response cookie(inName) → Text

Parameter	Type	Description
inName	Text	→ Name of response cookie to retrieve
Result	Text	← Value of response cookie or ""

Discussion

This command searches the response cookie collection for the item with the name *inName*.

If the item is found, the item’s value is returned.

If the item is not found, an empty string is returned.

Note: The cookie name and value are URL decoded and converted from UTF-8 to Unicode, so you do not need to perform any decoding.

get response cookies

version 2
modified v5

```
get response cookies(outNames {}; outValues{}
```

Parameter	Type	Description
outNames	Text Array	← Receives the cookie names
outValues	Text Array	← Receives the cookie values/attributes

Discussion

This command fills *outNames* and *outValues* with all of the names and values/attributes in the response cookies collection. Note that cookie attributes are stored along with the cookie value, as outlined in the Netscape cookie specification.

If *outNames* was not defined, it is created as a Text array. If *outValues* was not defined, it is created as a Text array.

If *outNames* is defined but is not a Text array, an error is generated an execution is aborted.

Note: The cookie names and values are URL decoded and converted from UTF-8 to Unicode, so you do not need to perform any decoding.

set response cookie

version 2
modified v6.1

```
set response cookie(inName; inValue {}; inExpires {}; inDomain {}; inPath {}; inSecure  
{}; inHttpOnly{}}{}}
```

Parameter	Type	Description
inName	Text	→ Name of cookie to set
inValue	<any>	→ Value to set
inExpires	Date/Timestamp	→ Expiration date/time of cookie
inDomain	Text	→ Cookie domain
inPath	Text	→ Cookie path
inSecure	Boolean	→ True to set Secure attribute
inHttpOnly	Boolean	→ True to set HttpOnly attribute

Discussion

This command sets a response cookie with the given attributes. *inValue* may be of any type; it is converted to text automatically. Because cookies must have a non-empty value, if the value is an empty string, it defaults to "null".

If the cookie already exists, its value and attributes are reset to those given.

If the item is not found, a new response cookie is added with the given name, value and attributes.

Note: Both *inName* and *inValue* are converted to UTF-8 and URL encoded, so you do not need to perform any encoding.

If *inExpires* is a date, the cookie will expire at the current time on that date. If *inExpires* is text which is a properly formatted timestamp, the cookie will expire at the date and time given in the timestamp. If *inExpires* is an empty string, it is ignored.

If *inPath* is not given or is empty, it defaults to `"/"`.

If *inSecure* is True and the current request is secure (https), the Secure attribute of the cookie will be set.

If *inHttpOnly* is True, the HttpOnly attribute of the cookie will be set.

For more information on the Secure and HttpOnly attributes, see http://en.wikipedia.org/wiki/Http_cookies#Secure_and_HttpOnly.

set response cookie domain

version 2

set response cookie domain(*inName*; *inDomain*)

Parameter	Type	Description
<i>inName</i>	Text	→ Name of cookie to set
<i>inDomain</i>	Text	→ Cookie domain

Discussion

This command sets the domain of the cookie with the given name.

You must create the cookie with **set response cookie** before using this command. If no cookie exists with the given name, an error is generated and execution is aborted.

get response cookie domain

version 2

get response cookie domain(inName) → Text

Parameter	Type	Description
inName	Text	→ Name of cookie to retrieve
Result	Text	← Domain of cookie

Discussion

This command returns the current domain of the cookie with the name *inName*. If no such cookie exists or the domain has not been set, an empty string is returned.

set response cookie expires

version 2

set response cookie expires(inName; inDate)

Parameter	Type	Description
inName	Text	→ Name of cookie to set
inDate	Date	→ Expiration date of cookie

Discussion

This command sets the expires date of the cookie with the name *inName*.

You must create the cookie with **set response cookie** before using this command. If no cookie exists with the given name, an error is generated and execution is aborted.

get response cookie expires

version 2

get response cookie expires(inName) → Date

Parameter	Type	Description
inName	Text	→ Name of cookie to retrieve
Result	Date	← Expiration date of cookie

Discussion

This command returns the current expires date of the cookie with the name *inName*. If no such cookie exists or the expires date has not been set, a null date is returned.

set response cookie http only

v6.1

```
set response cookie http only(inName; inHttpOnly)
```

Parameter	Type	Description
inName	Text	→ Name of cookie to set
inHttpOnly	Boolean	→ True to set HttpOnly attribute

Discussion

This command sets the HttpOnly attribute of the cookie with the given name.

You must create the cookie with **set response cookie** before using this command. If no cookie exists with the given name, an error is generated and execution is aborted.

get response cookie http only

v6.1

```
get response cookie http only(inName) → Boolean
```

Parameter	Type	Description
inName	Text	→ Name of cookie to retrieve
Result	Boolean	← HttpOnly attribute of cookie

Discussion

This command returns whether the HttpOnly attribute has been set for the cookie with the name *inName*. If no such cookie exists or the attribute has not been set, False is returned.

set response cookie path

version 2

```
set response cookie path(inName; inPath)
```

Parameter	Type	Description
inName	Text	→ Name of cookie to set
inPath	Text	→ Cookie path

Discussion

This command sets the path of the cookie with the given name.

You must create a cookie with **set response cookie** before using this command. If no cookie exists with the given name, an error is generated and execution is aborted.

get response cookie path

version 2

get response cookie path(inName) → Text

Parameter	Type	Description
inName	Text	→ Name of cookie to retrieve
Result	Text	← Path of cookie

Discussion

This command returns the current path of the cookie with the name *inName*. If no such cookie exists or the path has not been set, an empty string is returned.

set response cookie secure

v6.1

set response cookie secure(inName; inSecure)

Parameter	Type	Description
inName	Text	→ Name of cookie to set
inSecure	Boolean	→ True to set Secure attribute

Discussion

This command sets the Secure attribute of the cookie with the given name, but only if the current request is a secure (https) request.

You must create the cookie with **set response cookie** before using this command. If no cookie exists with the given name, an error is generated and execution is aborted.

get response cookie secure

v6.1

get response cookie secure(inName) → Boolean

Parameter	Type	Description
inName	Text	→ Name of cookie to retrieve
Result	Boolean	← Secure attribute of cookie

Discussion

This command returns whether the Secure attribute has been set for the cookie with the name *inName*. If no such cookie exists or the attribute has not been set, False is returned.

count response cookies

version 2

count response cookies → Longint

Parameter	Type	Description
Result	Longint	← Number of items in collection

Discussion

This command returns the number of items in the response cookies collection.

delete response cookie

version 2

delete response cookie(inName)

Parameter	Type	Description
inName	Text	→ Name of cookie to delete

Discussion

This command searches the response cookies collection for the item with the name *inName*. To delete more than one item, you may use a wildcard in the name. All items that match will be removed from the collection, not from the client.

abandon response cookie

version 2

abandon response cookie(inName {; inDomain {; inPath}})

Parameter	Type	Description
inName	Text	→ Name of cookie to abandon
inDomain	Text	→ Cookie domain
inPath	Text	→ Cookie path

Discussion

To delete a cookie from the client browser, you actually need to send a cookie with an expires date of !00/00/00!. This command is a convenience routine which does that for you.

If you change the *expires* attribute to something other than !00/00/00! after calling this command, the cookie will not be deleted from the client browser.

If the cookie to be abandoned was set with a domain and/or path, you must provide the same domain and/or path to successfully abandon it.

Response Headers

Active4D takes care of setting all of the response headers required by the HTTP protocol. In addition, it provides commands for setting various response properties that get turned into response headers (see “Response Properties” on page 324).

If you need to set your own custom headers, you can do so by putting them in the response header collection.

The response headers collection is read-write, and follows the same pattern as all read-write collections. All of the values in the response headers collection are text and are considered opaque by Active4D. It is up to you to do any necessary URL encoding or character set conversion.

response headers

version 2

response headers → Longint

Parameter	Type	Description
Result	Longint	← Iterator reference

Discussion

This command returns an iterator to the first item in the response headers collection.

For more information on iterators, see “Iterators” on page 214.

get response header

version 2

get response header(inName) → Text

Parameter	Type	Description
inName	Text	→ Name of response header to retrieve
Result	Text	← Value of response header or ""

Discussion

This command searches the response header collection for the item with the name *inName*.

If the item is found, the item’s value is returned.

If the item is not found, an empty string is returned.

get response headers

version 2

get response headers(outNames {}; outValues)

Parameter	Type	Description
outNames	String/Text Array	← Receives the header names
outValues	String/Text Array	← Receives the header values

Discussion

This command fills *outNames* and *outValues* with all of the names and values in the response headers collection.

If *outNames* was not defined, it is created as a string array. If *outValues* was not defined, it is created as a text array.

If *outNames* is defined but is not a string or text array, an error is generated an execution is aborted.

set response header

version 2

set response header(inName; inValue)

Parameter	Type	Description
inName	Text	→ Name of header to set
inValue	<any>	→ Value to set

Discussion

This command searches the response headers collection for the item with the name *inName*. *inValue* may be of any type; it is converted to text automatically.

If the item is found, its value is set to *inValue*.

If the item is not found, a new item is added to the collection with the given name and value.

count response headers

version 2

count response headers → Longint

Parameter	Type	Description
Result	Longint	← Number of items in collection

Discussion

This command returns the number of items in the response headers collection.

delete response header

version 2

delete response header(inName)

Parameter	Type	Description
inName	Text	→ Name of header to delete

Discussion

This command searches the response headers collection for the item with the name *inName*. To delete more than one item, you may use a wildcard in the name. All items that match will be removed from the collection.

Response Properties

Most of the standard HTTP response headers generated by Active4D can be controlled through the commands in this group.

get cache control

version 2

get cache control → Text

Parameter	Type	Description
Result	Text	← Current cache control setting

Discussion

This command returns the cache-control header setting.

set cache control

version 2

set cache control(inOption)

Parameter	Type	Description
inOption	Text	→ Cache control directive

Discussion

This command controls the “cache-control” response header. The default value for the cache-control response header is set by the “cache control” configuration option in Active4D.ini.

The value of *inOption* should conform to the HTTP caching protocol as outlined in RFC 2616. Since HTTP 1.0 clients do not recognize the cache-control header, if *inOption* is “no-cache”, Active4D also sends a “pragma: no-cache” header and an “expires” header with a time of now.

get expires

version 2

get expires → Longint

Parameter	Type	Description
Result	Longint	← Minutes till expiration

Discussion

This command returns the minutes until the response should expire.

set expires

version 2

```
set expires(inMinutes)
```

Parameter	Type	Description
inMinutes	Longint	→ Minutes till expiration

Discussion

This command controls the “expires” response header, which is used by HTTP clients to control caching of web pages. The default value for the expires response header is set by the “expires” configuration option in Active4D.ini.

inMinutes is the number of minutes before the response should expire. A value of zero forces the response to expire immediately. A negative value will cause this value to be ignored by Active4D. Positive values are clipped to one year (in minutes).

get expires date

version 2

```
get expires date(outDate {; outTime})
```

Parameter	Type	Description
outDate	Date	← Expiration date
outTime	Time	← Expiration time

Discussion

This command returns the date and time at which the response should expire.

set expires date

version 2

```
set expires date(inDate {; inTime})
```

Parameter	Type	Description
inDate	Date	→ Expiration date
inTime	Time	→ Expiration time

Discussion

As an alternative to setting the number of minutes till expiration, you may also specify the exact date and time at which the response should expire. The date/time specified is clipped to one year from now.

get content type

version 2

get content type → Text

Parameter	Type	Description
Result	Text	← MIME type of response

Discussion

This command returns the current MIME type of the response.

set content type

version 2

set content type(inType)

Parameter	Type	Description
inType	Text	→ MIME type of response

Discussion

This command sets the “Content-Type” header of the response, which should be a valid MIME type.

The default content type is “text/html”. Ordinarily you would have no need to change this, unless for example you are sending a plain text file, in which case you would set the content type to “text/plain”.

get content charset

version 2
deprecated v5

get content charset → Text

Parameter	Type	Description
Result	Text	← Charset name

Discussion

This command is deprecated and is now an alias for the **get output charset** command.

set content charset

version 2
deprecated v5

set content charset(inCharset)

Parameter	Type	Description
inCharset	Text Number	→ Charset name

Discussion

This command is deprecated and is now an alias for the **set output charset** command.

get response status

v6.0

get response status → Longint

Parameter	Type	Description
Result	Longint	← HTTP status

Discussion

This command returns the current HTTP status. Ordinarily the status is 200 (OK), unless you have changed it with the **set response status** command.

Within the context of an error handler, the response status is set to 200 (OK). If an error handler needs to get the response status that triggered it, use the **get error status** command.

set response status

version 2

set response status(inStatus)

Parameter	Type	Description
inStatus	Number	→ HTTP status code

Discussion

This command changes the status code that will be returned with the response. Named constants for the most common status codes can be found in Appendix B.

Script Environment

There are several configuration options that apply to Active4D's scripting environment as a whole. Many of those options can be changed at runtime with the commands in this section.

This section also contains utility commands for getting information about the host environment.

_request

version 4.0`_request` → Longint

Parameter	Type	Description
Result	Longint	← Request collection iterator

Discussion

This command returns an iterator to a special built in collection which is automatically created with each executable request and then is automatically cleared at the end of the request.

If you need to store data that is accessible globally, but only within a given request — as opposed to the **globals** collection, which is global to all requests — the **_request** collection is the perfect place to store that data.

full requested url

version 4.0`full requested url` → Text

Parameter	Type	Description
Result	Text	← Full path plus any query

Discussion

This command returns everything after the `hostname{:port}` that was part of the requested URL.

current platform

version 2`current platform` → Number

Parameter	Type	Description
Result	Number	← Current platform code

Discussion

This command returns a number representing the platform under which Active4D is running. The number will either be 2 or 3, which can be tested with the named constants *Power Macintosh* and *Windows* respectively.

In case you hadn't noticed, this is a convenient replacement for the standard 4D incantation:

```
C_LONGINT($platform)
PLATFORM PROPERTIES($platform)
$0:=$platform
```

get license info

version 2

```
get license info(outUserName; outCompany; outLicenseType; outLicenseVersion;
outServerIP; outExpirationDate; outPlatform {; outKeyFilePath})
```

Parameter	Type	Description
outUserName	Text	← The licensed user
outCompany	Text	← The licensed company
outLicenseType	Longint	← The type of license
outLicenseVersion	Text	← Active4D version licensed for
outServerIP	Text	← IP address for a regular deployment license, empty otherwise
outExpirationDate	Date	← Date a deployment license expires
outPlatform	Longint	← Always 3 (Mac and Windows)
outKeyFilePath	Text	← Full path to key file

Discussion

This command returns license information from the key file of the machine on which Active4D is running. The license types are as follows:

Type	License
0	Trial
1	Developer
2	Deployment
3	OEM
4	Special
5	Expired

If no key file is found, *outLicenseType* is 0 (Trial) and *outExpirationDate* is 00/00/00.

If *outLicenseType* is 3 (OEM), *outExpirationDate* is 00/00/00.

If *outKeyFilePath* is passed in, it receives the full URL path to the key file if it was found during startup.

get time remaining

version 2

get time remaining → Longint

Parameter	Type	Description
Result	Real	← Seconds till license timeout

Discussion

This command returns the number of seconds remaining until the current license times out.

If the license provides unlimited time, zero is returned. If the license has already timed out, -1 is returned.

get version

**version 1
modified v5**

get version → Text

Parameter	Type	Description
Result	Text	← Current Active4D version

Discussion

This command returns the following information about the instance of Active4D that is running:

Item	Values
version	Active4D 6.XrXX
architecture	Macintosh/Intel, Windows/x86
build type	debug, release

For example, the 6.0r1 release version of Active4D running on macOS would display the following version string:

```
Active4D 6.0r1 [Macintosh/Intel, release]
```

Note: The network layer, 4D host and build flags are no longer returned in v5+. These values can now be found in the Active4D log file, along with complete 4D and system information.

configuration

v6.0

configuration → Iterator

Parameter	Type	Description
Result	Iterator	← Configuration info collection

Discussion

This command returns an iterator to a read-only collection that contains the complete set of configuration information read from Active4D.ini, VirtualHosts.ini, Realms.ini, and ExtensionMap.ini.

The items of the collection are:

Item	Contents
cors	Array of collections, one for each entry in Cors.ini
extension map	Collection, key = extension, value = mime type
options	Collection, key = option name, value = option value
realms	Collection, key = realm name, value = match string
virtual hosts	Array of collections, one for each virtual host entry in VirtualHosts.ini

So, for example, if you want to retrieve the current set of virtual hosts, you would use this:

```
configuration{"virtual hosts"}
```

To dump the configuration to a web page, the easiest way is to use the `a4d.debug.dump` configuration method.

parameter mode

version 2

parameter mode → Text

Parameter	Type	Description
Result	Text	← Current parameter mode setting

Discussion

This command returns the current setting of the “parameter mode” option in Active4D.ini, which will be either “none”, “form variables” or “query params”.

request query

version 4.0`request query` → Text

Parameter	Type	Description
Result	Text	← Query string

Discussion

This command returns the query string portion of the requested URL, if any (without the leading "?").

set platform charset

**version 2
modified v5**`set platform charset(inCharset)`

Parameter	Type	Description
inCharset	Number Text	→ Charset to use when converting executable source files

Discussion

This command sets the character set from which executable source files are converted to Unicode in the current request.

You may either pass one of the character set constants below or an IANA character set name. The output character set constants supported by Active4D are:

Character Set	Constant
UTF-8	A4D Charset UTF8
Macintosh Roman	A4D Charset Mac
Windows Latin	A4D Charset Win
ISO-8859-1 (Latin1)	A4D Charset ISO Latin1
Shift_JIS	A4D Charset Shift_JIS
GB2312	A4D Charset GB2312

If you pass a name, it must be a valid IANA character set name. If the name is empty, it will default to "mac". If it is invalid an error will be generated and execution will abort.

For more information on the platform charset, see "Working with Character Sets" on page 104.

get platform charset

version 2
modified v5

get platform charset → Text

Parameter	Type	Description
Result	Text	← Name of the charset from which executable source files are converted

Discussion

This command returns the current platform charset, which determines how Active4D converts executable source files to Unicode.

The name returned by this command is the internal, canonical name used by ICU, and thus may not be the same name you used either in Active4D.ini or with the **set platform charset** command. For a complete list of charset names, see:

<http://demo.icu-project.org/icu-bin/convexp?s=IANA&s=ALL>

For more information on the platform charset, see “Working with Character Sets” on page 104.

set script timeout

version 2

set script timeout(inSeconds)

Parameter	Type	Description
inSeconds	Longint	→ Seconds script may run

Discussion

You use the script timeout to ensure that an errant script doesn't go into an infinite loop and tie up server resources indefinitely. Active4D checks the timeout before executing each line of code. If the script has been running more than *<script timeout>* seconds, an error is generated and execution is aborted.

This command sets the script timeout for the *next* execution of Active4D, not the one in which the command is used. In no case can the timeout be set lower than the “script timeout” setting in Active4D.ini.

get script timeout

version 2

get script timeout → Longint

Parameter	Type	Description
Result	Longint	← Seconds script may run

Discussion

This command returns the script timeout in seconds.

set current script timeout

version 2

set current script timeout(inSeconds)

Parameter	Type	Description
inSeconds	Longint	→ Seconds current script may run

Discussion

You use the script timeout to ensure that an errant script doesn't go into an infinite loop and tie up server resources indefinitely. Active4D checks the timeout before executing each line of code. If the script has been running more than *<script timeout>* seconds, an error is generated and execution is aborted.

This command sets the script timeout *only* for the *current* execution of Active4D. In no case can the timeout be set lower than the "script timeout" setting in Active4D.ini or the most recent execution of the *script timeout* command.

get current script timeout

version 2

get current script timeout → Longint

Parameter	Type	Description
Result	Longint	← Seconds script may run

Discussion

This command returns the timeout of the currently executing script in seconds.

Selecting Records

The commands in this section are enhanced versions of the standard 4D selection navigation commands. These commands are:

ALL RECORDS
FIRST RECORD
LAST RECORD
NEXT RECORD
PREVIOUS RECORD
GOTO RECORD
GOTO SELECTED RECORD

In addition to these commands, the **SELECTION TO ARRAY** and **SELECTION RANGE TO ARRAY** commands have been similarly enhanced. They are discussed in “SELECTION/SELECTION RANGE TO ARRAY” on page 150.

Loading Related Records

Each of the above commands has been enhanced to allow related records to be automatically loaded — after the record is selected — as if you had implicitly called the **RELATE ONE** or **RELATE MANY** commands. *This feature will work only with automatic relations.* More importantly, this feature is implemented directly by 4D, so it is very fast.

Note: The loading of related records spoken of here is distinct from the effects of the **AUTOMATIC RELATIONS** command, which only affects queries and order bys.

Configuring Related Record Auto-loading

There are three ways in which related records may be auto-loaded:

- **Globally:** Active4D.ini has a pair of settings called “auto relate one” and “auto relate many”. These settings determine the default behavior for the commands listed above. If you do not specify a value for these settings, the default is *False*. Setting them to *True* causes related one or related many records to be loaded whenever one of the commands above is executed, unless you override that behavior through one of the two techniques outlined below.
- **Per execution:** There is a new command, *auto relate(inRelateOne; inRelateMany)*, which sets auto-relating of records in the currently executing script. Whatever values you pass to this command will override the global setting from that point to the end of the script’s execution.
- **Per command:** Each of the commands listed above adds two optional boolean parameters, *inRelateOne* and *inRelateMany*. Specifying a value for these parameters overrides both the global default and the default for the currently executing script, if it was set by the *auto relate* command.

Compatibility with Active4D 2.0.x

Active4D 2.0.x always loaded related one and related many records for each of the commands above. Not only was this behavior in contradiction to standard 4D behavior, it could have adversely affected performance.

Beginning with version 3.0, the default behavior is the same as 4D's behavior: related one and related many records are *not* loaded when using the commands above. You must be sure that your code takes this into account.

Warning: It is quite possible that scripts in version 2.0.x relied on the undocumented behavior of the commands above and referenced values in related tables without using **RELATE ONE** or **RELATE MANY**. If such is the case, they must either add **RELATE ONE** or **RELATE MANY** commands or use one of the three techniques outlined above to automatically load related records.

Examples

Let's say you are looping through a selection of *[ingredients]* records and you want to display *[vendor]name*, which is related by a many to one relation from the *[ingredients]vendor_id* field to the *[vendors]id* field.

Here's how you would ordinarily do it in Active4D:

```
for ($i; 1; records in selection([ingredients]))
  goto selected record([ingredients]; $i)
  relate one([ingredients])
  writebr([ingredients]name + " comes from " + [vendors]name)
end for
```

If the global "auto relate one" setting is not set or is set to *False*, and you do not use the **auto relate** command, and you do not use the optional flags in the **GOTO SELECTED RECORD** command, this is how you must do it.

Now let's assume you want the convenience of always loading the related one records for all tables in your database. In that case you would change Active4D.ini:

```
auto relate one = true
```

By doing this, the code above could be written without using **RELATE ONE**:

```
for ($i; 1; records in selection([ingredients]))
  goto selected record([ingredients]; $i)
  writebr([ingredients]name + " comes from " + [vendors]name)
end for
```

Then you realize that there are many tables for which you don't want to load the related one record, so you change the "auto relate one" setting in Active4D.ini to be *False*. This

leaves you with two ways of loading the related *[vendor]* records without using **RELATE ONE**:

```
auto relate(true; false)
// From this point on auto-load related one

for ($i; 1; records in selection([ingredients]))
  goto selected record([ingredients]; $i)
  writebr([ingredients]name + " comes from " + [vendors]name)
end for
```

```
for ($i; 1; Records in selection([ingredients]))
  // passing extra argument
  goto selected record([ingredients]; $i; true)
  writebr([ingredients]name + " comes from " + [vendors]name)
end for
```

As you can see, there are many different ways to accomplish what you want. It is all a matter of how much control vs. convenience you want.

Here's one thing to watch out for:

```
auto relate(true)
query([ingredients]; [ingredients]name = "b@")
// First record is loaded but no auto relate happens

while(not(end selection([ingredients])))
  writebr([ingredients]name + " comes from " + [vendors]name)
end while
```

In this example, the first record in the selection resulting from the query would not have the related one record from *[vendors]* loaded, because no selection navigation commands were executed. To make this work correctly you would have to do this:

```
auto relate(true)
query([ingredients]; [ingredients]name = "b@")
first record([ingredients])
```

auto relate**version 3.0**

```
auto relate(inRelateOne {; inRelateMany})
```

Parameter	Type	Description
inRelateOne	Boolean	→ Sets auto-load of related one records
inRelateMany	Boolean	→ Sets auto-load of related many records

Discussion

This command sets the auto-loading of related one and related many records for the currently executing script. The settings specified here will override whatever the “auto relate one” and “auto relate many” settings are in Active4D.ini. The settings in this command will in turn be overridden by the extra parameters passed to the selection navigation commands as outlined below.

For a complete discussion of auto-loading of related records, see “Configuring Related Record Auto-loading” on page 337.

ALL RECORDS, FIRST/LAST/NEXT/PREVIOUS RECORD**version 3.0
modified v5**

```
ALL RECORDS([inTable] {; inRelateOne {; inRelateMany})
FIRST RECORD([inTable] {; inRelateOne {; inRelateMany})
LAST RECORD([inTable] {; inRelateOne {; inRelateMany})
NEXT RECORD([inTable] {; inRelateOne {; inRelateMany})
PREVIOUS RECORD([inTable] {; inRelateOne {; inRelateMany})
```

Parameter	Type	Description
inTable	Table	→ Table on which to act
inRelateOne	Boolean	→ Auto-load related one records
inRelateMany	Boolean	→ Auto-load related many records

Discussion

These commands act on the current record of *inTable* exactly as they do in 4D. In addition, if *inRelateOne* and/or *inRelateMany* are passed, they do the following:

- **inRelateOne:** After the current record in *inTable* has been changed, if this parameter evaluates to *True*, the related one record is loaded for each many to one relation in *inTable*. The relations *must* be automatic for this to work.

If this parameter evaluates to *False*, the related one records will not be loaded. If this parameter is not passed, the behavior is specified by the *auto relate* command or the “auto relate one” setting in Active4D.ini.

- **inRelateMany:** The same as *inRelateOne*, but this controls the loading of related many records. If not passed the default behavior is specified by the *auto relate* command or the “auto relate many” setting in Active4D.ini.

get auto relations

version 3.0

get auto relations(outRelateOne; outRelateMany)

Parameter	Type	Description
outRelateOne	Boolean	← Auto-loading of related one records
outRelateMany	Boolean	← Auto-loading of related many records

Discussion

This command gets the current state of auto-loading of related one and related many records for the currently executing script.

For a complete discussion of auto-loading of related records, see the command “auto relate” on page 340 and “Configuring Related Record Auto-loading” on page 337.

GOTO RECORD

(modified 4D) version 3.0

GOTO RECORD([inTable]; inRecordNum {; inRelateOne {; inRelateMany{}}

Parameter	Type	Description
inTable	Table	→ Table on which to act
inRecordNum	Number	→ Number of record to load
inRelateOne	Boolean	→ Auto-load related one records
inRelateMany	Boolean	→ Auto-load related many records

Discussion

This command acts on the current selection and current record of *inTable* exactly as it does in 4D. In addition, the *inRelateOne* and *inRelateMany* parameters act as they do in the **FIRST RECORD** command, as described above.

GOTO SELECTED RECORD**(modified 4D) version 3.0**

GOTO SELECTED RECORD([inTable]; inOffset {; inRelateOne {; inRelateMany{}})

Parameter	Type	Description
inTable	Table	→ Table on which to act
inOffset	Number	→ Offset within selection to make current
inRelateOne	Boolean	→ Auto-load related one records
inRelateMany	Boolean	→ Auto-load related many records

Discussion

This command acts on the current record of *inTable* exactly as it does in 4D. In addition, the *inRelateOne* and *inRelateMany* parameters act as they do in the **FIRST RECORD** command, as described above.

Sessions

In 4D, between the time you start a process and the time the process dies, you can maintain the state of the process by using process variables.

The HTTP protocol, on the other hand, is completely stateless. Between the time a client requests one page and another, the HTTP protocol provides no way of maintaining persistent state information about the client.

Client-side cookies are one way of storing persistent information, but they are cumbersome and inefficient for storing much more than a few pieces of information. Thus the need arises for providing a persistent state mechanism.

The solution which is widely accepted by web scripting environments is called a *session*. A session is a read-write collection which is maintained on the server and can be used to store and retrieve any number of values pertaining to a client's state, much as you would use process variables. Active4D transparently does all the work of creating and maintaining sessions.

If you have never programmed for the web before, you will quickly discover that good session support is worth its weight in gold. If you have already programmed for the web, you know that good session support is very difficult — meaning very expensive — to implement. If you ask anyone who has ever tried to implement session support themselves, they will tell you that Active4D's session support alone makes it worth the price!

Active4D's session implementation follows the session architecture used in industry-leading web scripting engines such as PHP, JSP, and ASP. This relieves you of the considerable burden of implementing world-class session support yourself.

Session Lifetime

Active4D maintains a client's session data for the lifetime of the session. The lifetime of a session is defined by two factors:

- Your scripts may specifically use the **abandon session** command to expire a session. Typically you would do this when a client logs out of your system.
- If you do not specifically abandon the session, the session's lifetime is defined by its *timeout*, which is the maximum amount of time *between requests* that a session will stay alive.

A session is still valid in code that is executed within the request in which it is abandoned or expires, however any changes made to the session will be lost. In subsequent requests the session's id is no longer valid.

What happens to a session after it is abandoned or expires depends on which session handler is in use. For more information on session handlers, see "Session Handlers" on page 348.

- **Built-in (memory):** An abandoned or expired session is not immediately deleted from memory. Its time remaining is set to zero so that it will be deleted in the next cycle of the session purger.

- **Custom:** At the end of the request, the “delete” method is called if it is defined, and that method should delete the session’s storage. Expired sessions should be deleted in the “purge” method if it is defined.

A background process periodically checks all sessions to see if they have timed out and purges those that have. If you are using the built-in memory-based session handler, the **On Session End** event handler is called for each session that is about to be purged. Within the context of that handler, the about-to-be-purged session is current and you can access all of its data.

Note: If you are using a custom session handler, **On Session End** is not called.

By default the timeout for a session is 10 minutes. This means that if a user logs in to your site, does some operations, and then leaves your site without logging out, their session will remain in memory for an additional 10 minutes before being deleted by the session purger.

Session ID

When a client visits your web site, Active4D looks for a cookie called “ACTIVE4D_SESSIONID”. If the cookie is not there, Active4D assumes this is the first time the client has visited your site since they launched their browser.

Active4D then generates a new *internal session ID* and *public session ID*. Since you only work with the public session ID, it is simply called the *session ID*. Internal and public session IDs have the following characteristics:

- Internal session IDs are 32-bit unsigned longints.
- Public session IDs are 16-character strings.
- If you are using the built-in memory-based session handler, each time 4D is started, a random internal session ID starting value is selected. Then for each Active4D session that is created, the internal session ID is incremented.
- If you are using a custom session handler, the handler is responsible for generating unique internal session IDs.
- The 32-bit internal session ID is mixed with random data and encrypted to generate a 16-character cookie string.
- The encryption key is randomly selected each time the 4D is restarted.

The client never sees the internal session ID, only the encrypted session ID, unless you specifically include the internal ID on your page with the **session internal id** command.

Note: For your internal use, you should always use the internal session id.

Session Events

When Active4D generates a new session ID, it checks to see if an **On Session Start** event handler has been defined in the Active4D library. If such an event handler is defined, it is executed.

If you are using the built-in memory-based session handler, when the session is deleted Active4D executes the **On Session End** event handler if one exists. For more information on event handlers, see Chapter 10, “Event Handlers.”

When Active4D Sends Session Cookies

Even though Active4D generates a session ID when a client accesses your site, it does not create the session in memory or send a session cookie to the client unless when one of the following happens:

- The **On Session Start** event handler is defined (it need not set any session data)
- A script sets some session data using a **set session** command or by assigning a value to a session item
- A script uses the **session** command without referencing an item

Note: If the session cookie secure configuration option is *true* and the request is not secure, the session cookie will not be sent even if one of the above conditions is met.

If one of these conditions is not met, the next time that client browser accesses the server, a new session ID will be generated because there will be no session cookie.

On the other hand, if the **On Session Start** event handler is defined or a session item is set, Active4D will send a cookie called “ACTIVE4D_SESSIONID” (by default) back to the client. The cookie’s value is the 16-character session ID.

Each subsequent request by that browser — until the client restarts the browser — will include the session cookie with the session ID. Active4D decrypts the session ID into the internal session ID and restores the corresponding session collection. All of the data stored in the session during previous requests is then available to the current script.

The session cookie sent by Active4D does not have an expiration. Cookies with no expiration are deleted when the browser closes, so a session is only valid as long as the originating browser remains open.

Note: Active4D uses session cookies by default. You may disable session cookies (and session management as well) through the config options in Active4D.ini. For more information, see “Session Configuration” on page 347.

Cookieless Sessions

If you decide to forgo cookies as a means of storing the session ID, *it is your responsibility to ensure that the session ID is always passed from one page to another*, either through a hidden form field or through a query string parameter. If you fail to do so, you will lose track of the session.

Active4D automatically creates a local variable which contains the session ID. The name of this variable is “sid” by default; you can change it with the “session var name” option in Active4D.ini.

If Active4D is configured to use session cookies and it cannot find the cookie, or if it is configured not to use cookies, it automatically looks for the session variable either in the query params or in the form variables and then retrieves the session ID from the contents of that variable.

When you want to pass the session ID as a hidden field, use the **hide session field** command. To pass the session ID in a query string, use the **session query** command to get a properly formatted query parameter.

Memory Caching of Sessions

By default, Active4D caches sessions in memory. Sessions use a minimum of 260 bytes of memory per item. Thus if you have on average a session with 20 items, that means you will use somewhere in the neighborhood of 5K. If you expect to have 100 sessions alive at any given time, that means the session cache will occupy 500K of memory. This will give you a rule of thumb in terms of how much memory usage to allocate for.

Because of the amount of memory sessions use, you should use sessions judiciously in your application if you are not using a custom session handler. You would not want to store huge numbers of items in a session, nor store huge amounts of text in a session item. If this is necessary, consider either caching sessions in the database or storing some info in the database and some in the session.

Session Timeout and Memory Usage

If you are not using a custom session handler, the maximum amount of memory sessions will use in your web site can be expressed with the formula

$$M = V \times T \times S$$

where M is the total memory usage, V is the number of unique visitors per minute, T is the session timeout, and S is the average memory usage per session.

So, for example, if you have a session timeout of 10 minutes, 10 unique visitors per minute, and an average of 5K per session, your maximum memory usage will be 500K.

You can use this formula to tune the session timeout. Remember that the whole purpose of sessions is to provide continuity in the user's state while using your site. Therefore you typically do not want the timeout to be less than 5 minutes, as the user will lose their session if they stop to get a cup of coffee! On the other hand, you don't want the session timeout to be too high because you will unnecessarily be wasting server memory. At some point you have to assume the user has stopped using your site.

Monitoring Memory Usage

Active4D provides a 4D-based session monitor that displays the total memory usage for all sessions. Using this during a typical load on your server can help you to tune your timeout and the amount of memory you give to your server.

For more information on the session monitor, see “The Session Monitor” on page 604.

Note: If you are using a custom session handler, the session monitor will show any sessions.

Session Configuration

Active4D lets you configure every aspect of session management through options in Active4D.ini.

session handler

If set, a custom session handler will be used instead of the built-in memory-based session handler. For more information, see “Session Handlers” on page 348.

session cookie domain

HTTP cookies have a domain associated with them. The browser uses the domain and path to determine whether to send a cookie to the server.

You may set the default session domain with this option. Ordinarily you would have no need to do so.

The default value for this option is to have no domain.

session cookie name

This option determines the name of the cookie which Active4D looks for to find a session ID when the user makes a request. It must contain only alphanumeric characters or underscores with no spaces.

The default value for this option is “ACTIVE4D_SESSIONID”.

session cookie path

HTTP cookies have a virtual path associated with them. The browser uses the domain and path to determine whether to send a cookie to the server.

You may set the default session path with this option. Ordinarily you would have no need to do so.

The default value for this option is to have no path.

session cookie secure

If you want session cookies to be sent only when the request is secure (https), set this option to *true*.

The default value for this option is *false*.

session purge interval

This option determines the minimum time in seconds between attempts to purge expired sessions. In actual fact the interval may be something greater, since Active4D waits for all requests to finish before purging. Valid values for this option are between 5 and 60 inclusive.

The default value for this option is 10.

session timeout

The length of time in minutes that a session can live without any user interaction. Values less than 1 will be ignored. The session timeout can be changed at runtime with the **set session timeout** command. To set the timeout to a value less than one minute, use a fraction. For example, a value of 0.5 would be 30 seconds.

The default value for this option is 10.

session var name

The name of the local variable to automatically set to the current session ID (with or without the leading '\$'). It must contain only alphanumeric characters or underscores with no spaces.

The default value for this option is "sid".

use session cookies

This option determines whether Active4D should store session IDs in cookies or look for the session variable. Setting this option to "true" or "yes" will turn cookies on, "false" or "no" will turn them off.

The default value for this option is "true".

use sessions

This option is a global switch for session management. If you are using sessions for your application, specify "true" or "yes". If you do not need session management at all, specify "false" or "no".

The default value for this option is "true".

Session Handlers

By default, all sessions are kept in server memory. This is fast and convenient, but if you are using multiple servers with a load balancer, you will need to keep the sessions in globally accessible persistent storage, such as the database. The easiest way to do so is to install the database session handler Active4D provides. For information on how to install it, see "Installing the Predefined Session Handler" on page 35.

A session handler is either a group of 4D methods with a common name prefix, or an Active4D library. Session handlers contain the following methods:

- **nextId:** Returns the next internal id for new sessions.
- **read:** Returns session data from persistent storage given an id and expire time.
- **write:** Writes session data to persistent storage.
- **delete:** Deletes persistent storage for a session given an id.
- **purge:** Deletes all sessions in persistent storage which expired before a given time.

Of these methods, **nextId**, **read**, and **write** are mandatory. If one or more of them are not defined, the session handler is considered invalid and session support is turned off. The delete and purge methods are optional in case you wish to implement your own garbage collection mechanism.

For more information on session handler methods, see “SessionHandler” on page 594.

A custom session handler is activated by setting the “session handler” configuration option. To use 4D methods for the session handler methods, specify the base name of the 4D methods in the “session handler” configuration option. For example, if you specify “sessionHandler” (without the quotes), Active4D will check for the following methods:

```
sessionHandlerNextId  
sessionHandlerRead  
sessionHandlerWrite  
sessionHandlerDelete  
sessionHandlerPurge
```

To use an Active4D library as the session handler, specify the library name (without extension) in the “session handler” configuration option, prefixed with “*”. For example, if you specify “*sessionHandler” (without the quotes), Active4D will attempt to load the “sessionHandler” library and then check the library for the following methods:

```
nextId  
read  
write  
delete  
purge
```

Note: For either type of session handler, case is ignored both in the configuration option and in the method names.

Active4D does not check the session handler methods until all of your configuration options have been read. This ensures in particular that the “lib dirs” option is active when searching for a session handler library, which enables you to put it in one of the “lib dirs” directories.

session**version 2**
modified v6.0

session → Longint

Parameter	Type	Description
Result	Longint	← Iterator reference

Discussion

This command returns an iterator to the first item in the current session.

Note: If used by itself without referencing an item, this command will create a session if one does not exist.

For more information on iterators, see “Iterators” on page 214.

session to blob**version 3.0**
modified v5.0

session to blob{(ioBlob)} → BLOB | <none>

Parameter	Type	Description
ioBlob	BLOB	↔ BLOB to append data to
Result	BLOB	← Serialized session data

Discussion

This command serializes the data in the current session. If *ioBlob* is passed, the serialized data is appended to *ioBlob* and nothing is returned. If *ioBlob* is not passed, the serialized data is returned as a new BLOB. In either case, you can store the BLOB somewhere for later restoral via **blob to session**.

For an example of how to serialize and deserialize a session with embedded collections, see “collection to blob” on page 156.

blob to session

version 3.0
modified v5.0

blob to session(inBLOB {; ioOffset})

Parameter	Type	Description
inBLOB	BLOB	→ BLOB with serialized collection data
ioOffset	Number	↔ Offset within BLOB to get data

Discussion

This command sets the current session from the serialized collection data contained in *inBLOB*. If *ioOffset* is passed, the serialized data must begin at that byte offset within *inBLOB*. After the session is successfully deserialized, *ioOffset* will point to the first byte beyond the serialized data.

If *inBLOB* was not created with **session to blob**, an error will be generated and execution will be aborted.

Note that this command will run the **On Session End** event handler on the old session, but the **On Session Start** event handler will not be run for the restored session, since you don't want to initialize it. Depending on what you do in **On Session Start**, you may have to factor out some code into a separate method and then call that method both from **On Session Start** and after calling **blob to session**.

For an example of how to serialize and deserialize a session with embedded collections, see "collection to blob" on page 156.

get session

version 2

get session(inKey {; inIndex}) → <any>

Parameter	Type	Description
inKey	Text	→ Key of session item to retrieve
inIndex	Number	→ Array element to retrieve
Result	<any>	← Value of session item or ""

Discussion

This command searches the current session for the item with the key *inKey*.

If the item is found and *inIndex* is not specified, the item's value is returned. If the item value is an array, an empty string is returned.

If the item is found and an index is specified, the given array element is returned. If the index is out of range, an error is generated and execution is aborted.

If the item is not found, an empty string is returned.

Note: This command has been superceded by the simpler syntax:

`session{inKey}` or `session{inKey}{inIndex}`

get session array

version 2

get session array(inKey; outArray)

Parameter	Type	Description
inKey	Text	→ Key of session item to retrieve
outArray	Array	← Receives the array

Discussion

This command searches the current session for the item with the key *inKey*.

If *outArray* is not an array, an error is generated and execution is aborted.

If the item is found and its value is an array, *outArray* receives a copy. If *outArray* has not yet been defined, it is created with the same type as the source array.

If the item is found and its value is not an array, an error is generated and execution is aborted.

If the item is not found, *outArray* is resized to zero. If *outArray* has not yet been defined, it is created as a text array.

Note: Because you can reference arrays directly with the syntax `session{"key"}`, you may apply any array commands directly to the array within the session. Thus there is no longer any need for this command. It is kept only for backward compatibility.

get session array size

version 2

get session array size(inKey) → Longint

Parameter	Type	Description
inKey	Text	→ Key of session item to check
Result	Longint	← Size of array

Discussion

This command searches the current session for the item with the key *inKey*.

If the item is found and its value is an array, the size of the array is returned.

If the item is found and its value is not an array, an error is generated and execution is aborted.

If the item is not found, zero is returned.

Note: This command has been superseded by the simpler syntax:
size of array(session{inKey})

get session item

version 2

get session item(inKey) → Longint

Parameter	Type	Description
inKey	Text	→ Key of session item to retrieve
Result	Longint	← Iterator for session

Discussion

This command searches the current session for the item with the key *inKey*.

If the item is found, an iterator reference for the item is returned.

If the item is not found, an empty iterator is returned. For information on empty iterators, see “Iterating Over a Collection” on page 102.

get session keys

version 2

get session keys(outKeys)

Parameter	Type	Description
outKeys	String/Text Array	← Receives the session keys

Discussion

This command fills *outKeys* with all of the keys in the session. If *outKeys* has not yet been defined, it is created as a string array.

If *outKeys* is defined but is not a string or text array, an error is generated and execution is aborted.

set session**version 2**

```
set session(inKey; inValue {; inKeyN; inValueN | inIndex})
```

Parameter	Type	Description
inKey	Text	→ Key of session item to store
inValue	<any>	→ Value to set for the given item
inIndex	Longint	→ Index of array element to set

Discussion

This command searches the current session for the item with the key *inKey*.

If the item is found and its value is not an array, you may pass more than one key/value pair to set multiple items at once. If the item is found and its value is an array, you may pass an index to set an element of the array.

If the item is found and an index is not specified, the item's value is replaced with *inValue*.

If the item is found, its value is an array, and an index is specified, the given array element is set. If the index is out of range or the type of *inValue* is not assignment-compatible with the array, an error is generated and execution is aborted.

If the item is not found and an index is not specified, a new item is added to the session with the given keys and values.

If the item is not found and an index is specified, an error is generated and execution is aborted.

Note: This command has been superseded by the simpler syntax:

```
session{inKey} := inValue or session{inKey}{inIndex} := inValue
```

Here is an example of **set session** using multiple items:

```
set session("name"; "Aparajita"; "age"; 40)
// session now contains two items
```

set session array

version 2

```
set session array(inKey; inArray)
```

Parameter	Type	Description
inKey	Text	→ Key of session item to store
inArray	Array	→ Array to set for the given item

Discussion

This command searches the current session for the item with the key *inKey*.

If *inArray* is not an array, an error is generated and execution is aborted.

If the item is found, its value is replaced with *inArray*.

If the item is not found, a new item is added to the session with the given key and array.

Note: You may directly declare an array within a session using the syntax:

```
array <type>(session{"key"})
```

However, *before* using this syntax you must make sure to either:

- Assign a value to a session item
- Use the **session** command by itself
- Use one of the **set session** commands

session has

version 2

```
session has(inKey {; *}) → Boolean
```

Parameter	Type	Description
inKey	Text	→ Key of session item to test
*	*	→ Perform wildcard search
Result	Boolean	← True if key is in session

Discussion

This command searches the current session for the item with the key *inKey*. If * is passed, *inKey* may contain 4D wildcard characters and they will be honored in the search.

If the item is found, *True* is returned, otherwise *False*.

Because a session value can be of any type, it is not sufficient to use `session{item}` and check for an empty string being returned. You should always use this command to test for the existence of a session item, as shown here:

```
if (session{"test"} # "")
  // The test above would break if the item "test" existed
  // and was a number
end if

// The correct way
if (session has("test"))
  $test := session{"test"}
else
  session{"test"} := "this is a test"
end if
```

count session items

version 2

count session items → Longint

Parameter	Type	Description
Result	Longint	← Number of items in session

Discussion

This command returns the number of items in the current session.

delete session item

version 2

delete session item(inKey)

Parameter	Type	Description
inKey	Text	→ Key of session item to delete

Discussion

This command searches the current session for the item with the key *inKey*. To delete more than one item, you may use a wildcard in the key. All items that match will be removed from the session.

abandon session

version 2

abandon session

Discussion

This command marks the current session to be expired when the current script finishes executing. After this command is executed, any changes made to the session are effectively lost.

An abandoned session is not removed from memory and the *On Session End* handler is not run until the next session purge cycle.

session id

version 2

session id → Text

Parameter	Type	Description
Result	Text	← Encrypted public session ID

Discussion

This command returns the 16-character encrypted public session ID, not the internal session ID.

session internal id

version 2

session internal id → Longint

Parameter	Type	Description
Result	Longint	← Internal session ID

Discussion

Active4D never directly uses the internal session ID. This command is here primarily because the Active4D Session Monitor (see “The Session Monitor” on page 604) displays internal session IDs. By using this command you can identify which session is active for a given browser and then view it in the Session Monitor.

session local

version 2

session local → Text

Parameter	Type	Description
Result	Text	← Name of the session local variable

Discussion

This command returns the name of the session ID local variable set by the “session var name” option in Active4D.ini.

session query

version 2

session query → Text

Parameter	Type	Description
Result	Text	← String suitable for using in a query string

Discussion

This command is a convenience routine that returns the equivalent of:

```
session local + "=" + session id
```

If you are using cookie-based sessions, you would ordinarily have no use for this command. But if you are not using cookie-based sessions, it is your responsibility to pass the session ID in every form and every link.

This command makes it easy to generate the correct query string parameter when building your links.

hide session field

version 2

hide session field

Discussion

This command is a convenience routine that performs the equivalent of:

```
write("<input type='hidden' name='%1%' id='%1%'
      value='%2%' />" % (session local; session id))
```

If you are using cookie-based sessions, you would ordinarily have no use for this command. But if you are not using cookie-based sessions, it is your responsibility to pass the session ID in every form and every link.

This command makes it easy to generate the correct hidden field within a form.

Note: Make sure this command is called inside a form, otherwise it will have no effect.

set session timeout

version 2

set session timeout(inMinutes)

Parameter	Type	Description
inMinutes	Real	→ Maximum time to live between requests

Discussion

This command sets the timeout of the current session in minutes. You can pass a fraction to set the timeout to less than one minute. For example, a value of 0.5 means 30 seconds.

get session timeout

version 2

get session timeout → Real

Parameter	Type	Description
Result	Real	← Maximum time to live between requests

Discussion

This command returns the timeout of the current session in minutes.

get session stats

version 3.0

get session stats(outIDs; outTimeouts; outRemainTimes; outSizes)

Parameter	Type	Description
outIDs	Longint Array	← Receives internal session IDs
outTimeouts	Real Array	← Receives session timeouts in minutes
outRemainTimes	Longint Array	← Receives seconds remaining till sessions timeout
outSizes	Longint Array	← Receives the memory used by the sessions in bytes

Discussion

This command retrieves information about all current sessions. You can use this information to create an interactive web-based monitor of all current sessions.

The arrays need not have been declared before calling this command, as they will be created on the fly with the specified type. If the array does exist, its type will be changed to the specified type.

Sessions which have expired but have not yet been purged will be returned with a remaining time of zero.

Note: This command is used by the a4d.debug library method **dump session stats**. For more on this method, see “dump session stats” on page 444.

Strings

Many, many new and incredibly useful string utilities have been added in Active4D. Learn them and you will save enormous amounts of time. Many of the most important additions to string handling are not in the form of commands but in the form of enhancements to the language. These are covered in the “Interpreter” chapter .

URL Encoding/Decoding

Internet standards dictate that URLs passed to web servers contain only a certain subset of the US-ASCII character set. All other characters are to be encoded in the form `%NN`, where `NN` is the hexadecimal value of the character in the US-ASCII character set.

Active4D automatically decodes the standard HTTP request elements such as header names, form variables, query string parameters, etc., and converts them from UTF-8 to Unicode. However, there are some request elements, such as cookie values, that are considered opaque and are not decoded.

Active4D provides a set of commands that facilitate the encoding and decoding of such opaque data:

```
url decode  
url decode path  
url decode query  
url encode  
url encode path  
url encode query
```

The “query” commands differ from the “path” commands in that:

- The “path” commands pass through more characters unencoded (such as `'/'`) and convert spaces to and from `%20`.
- The “query” commands encode everything but `-_~!*'()` and convert spaces to and from `'+'`.

You should use the “path” commands on the path portion of a URL, and the “query” commands on the query string portion of a URL.

When encoding, Unicode text is converted to UTF-8 and then encoded.

String Commands and Unicode

Internally Unicode text is stored using 16-bit (2-byte) code units. A single Unicode code point (what we would usually call a character) may require two code units (four bytes). However, almost all modern languages are in the Unicode Basic Multilingual Plane (or BMP), which can be represented by a single 16-bit code unit. When referring to strings in Active4D (and in 4D v11+), all indexes, lengths, etc. are in terms of 16-bit code units. Since any language you are likely to use will be in 16-bit code units, you should not have to worry about a character index falling in the middle of a 32-bit code point.

If the preceding paragraph made no sense to you, don't worry. It is unlikely you will ever have to understand what it means.

% (formatting operator)

version 4.0
modified v5

<format> % (arg1 {; ... argN}) → Text

Parameter	Type	Description
format	Text	→ Format string
arg1...N	<any>	→ Format arguments
Operator result	Text	← Formatted text

Discussion

This operator takes a format string and applies one or more arguments to it. The format string can be any mixture of literal text and formatting directives which indicate the format and type of the argument that it applies to.

Note: A BLOB argument is treated as raw text stored as *UTF8 Text without length*.

Note that parameter numbers are one-based with the % operator.

The full syntax of the format string is beyond the scope of this document. You can read its documentation here:

<http://www.boost.org/libs/format/doc/format.html#syntax>

Example

```
// $cart is a RowSet
$i := 1
$format := "%1%. %2% $%3$4.2f"

while ($cart->next)
  $item := $cart->getRow
  writebr($format % ($i++; $item{"desc"}; $item{"price"}))
end while

// output
1. Acme Jet-pack $123.45
2. Spacely sprocket $27.95
3. Cogswell cog $7.99
```

There are many, many online examples of how to use this type of formatting. Just enter "printf examples" or "printf tutorial" into your favorite search engine.

Note: In general the %% operator should be preferred over % because it works directly with Unicode, whereas the % operator has to convert all arguments to UTF-8, perform the formatting, then convert the result back to Unicode.

%% (formatting operator)

v5
modified v6.0r10

```
<format> %% (arg0 {; ... argN}) → Text
```

Parameter	Type	Description
format	Text	→ Format string
arg0...N	<any>	→ Format arguments
Operator result	Text	← Formatted text

Discussion

This operator takes a format string and applies one or more arguments to it. The format string can be any mixture of literal text and formatting directives which indicate the format and type of the argument that it applies to.

Note: A BLOB argument is treated as raw text stored as *UTF8 Text without length*.

Active4D uses Internet Components for Unicode (ICU) to do formatting. There are many options available; ICU formatting is extremely powerful. The full syntax of the format string is beyond the scope of this document. To find out more, read the “Detailed Description” section of this page:

<http://icu-project.org/apiref/icu4c/classMessageFormat.html>

Active4D supports longints, doubles, strings, times, booleans, dates and collections as format values.

Booleans are converted to the number 1 or 0, which is useful with choice formats.

Using a collection as a format value allows you to refer to a collection value by name in the format string, where the name is one of the collection key. You may freely mix positional format values (non-collections) and named format values (collections) in the format value list. See the example below for an idea of how this works.

Format placeholders without any type specifier (such as “{0}” or “{name}”) convert their values by effectively using the **String** command. Thus the formatting will follow 4D’s default formatting, which is locale aware. Format placeholders with a type specifier (such as “{0,number}” or “{birthdate,date,long}”) are converted using the default formatting of the ICU locale, which is set to the system locale. Note that ICU’s locale formatting may not match 4D’s formatting exactly.

With the %% operator, unlike with the % operator, parameter numbers are zero-based.

The **format string** command is the equivalent of the %% operator in command form.

Note: In general the %% operator should be preferred over % because it works directly with Unicode, whereas the % operator has to convert all arguments to UTF-8, perform the formatting, then convert the result back to Unicode.

Example

```

// ¤ is the ICU format character for the localized currency symbol
$format := "{0}. {desc} {price, number, ¤###0.00}{2}"
$i := 1

// $cart is a RowSet with "desc" and "price" items.
// getRow() returns a collection with the current row's values.
$row := $cart->getRow

while ($cart->next)
    $special := choose($i = 2; " Special!"; "")
    writebr($format %% ($i++; $row; $special))
    // note that $special is used in the format as argument 2
end while

// output
1. Acme Jet-pack $123.45
2. Spacely sprocket $27.95 Special!
3. Cogswell cog $7.99

```

capitalize**version 3.0**

capitalize(*inText* {; *}) → Text

Parameter	Type	Description
<i>inText</i>	Text	→ Text to capitalize
*	*	→ Affect first letter only
Result	Text	← Capitalized text

Discussion

This command capitalizes the first letter of each word it finds in *inText*. Word boundaries are defined by punctuation, control characters and whitespace, excluding single quotes (they are treated as apostrophes) and underscores.

If * is not passed in, letters after the first letter are lowercased. If * is passed in, they are skipped. This allows you to maintain capitalization of words like "ID".

```

write(capitalize("THIS IS ALL CAPS"))
// output is "This Is All Caps"

write(capitalize("name, ID"; *))
// output is "Name, ID"

```

cell**version 3.0**cell(*inValue*) → Text

Parameter	Type	Description
<i>inValue</i>	<any>	→ Value to convert to text
Result	Text	← Converted value or " "

Discussion

Because some browsers (notably Navigator 4.x) do not properly draw cells if they have no contents, it is best to fill empty cells with non-breaking spaces. You can use the **cell** command to quickly and easily take care of this without having to check for empty contents yourself.

This command attempts to convert *inValue* to text according to the rules used by the **String** command. If *inValue* can successfully be converted to text, the length of the resulting text is checked.

If the length is zero, the command returns the HTML non-breaking space for the current output charset, suitable for placing in the cell of an HTML table.

If the length is non-zero, the converted text is returned.

Example

```
<tr>
  <td><% =cell([People]Fax) %></td>
</tr>
```

Note: This command is not necessary in modern browsers. The same effect can be achieved at a global level by putting the following style declaration in your main style sheet:

```
table: { border-collapse: collapse }
td, th: { empty-cells: show }
```

compare strings

version 2

compare strings(inSource; inCompare) → Longint

Parameter	Type	Description
inSource	Text	→ Text to compare with
inCompare	Text	→ Text to compare against
Result	Longint	← Relative result of comparison

Discussion

This command performs a bitwise comparison of the bytes in two strings. If *inSource* is bitwise greater than *inCompare*, a positive number is returned. If *inSource* and *inCompare* are bitwise equal, zero is returned. If *inSource* is bitwise less than *inCompare*, a negative number is returned.

concat**version 2**

concat(inDelimiter; inString1; inString2 {; ...inStringN}) → Text

Parameter	Type	Description
inDelimiter	Text	→ Text to put between strings
inString1	Text	→ First string
inString2	Text	→ Second string
Result	Text	← Concatenation of all strings

Discussion

This command concatenates two or more strings together, placing *inDelimiter* between them. If a string is empty, no delimiter is inserted.

This is basically a convenience routine that makes it easy to build full names, addresses, etc. without inserting extra spaces.

For example:

```
writebr("[ " + concat(" "; "John"; "J."; "Smith") + "]")
writebr("[ " + concat(" "; "John"; " "; "Smith") + "]")

// output
[John J. Smith]
[John Smith]
```

Delete string**version 1
modified version 4.0**

Delete string(inString; inFirstChar {; inNumChars}) → Text

Parameter	Type	Description
inString	Text	→ String from which to delete
inFirstChar	Number	→ Position of first character to delete
inNumChars	Number	→ Number of characters to delete
Result	Text	← Remaining string

Discussion

The Active4D version of the **Delete string** command differs from 4D's version in that you can pass a negative number for *inFirstChar* and *inNumChars*, which is converted to the index of the *Nth* Unicode code unit from the end of *inString*.

When a negative number is passed for *inNumChars*, note that it signifies an index relative to the end of the string, not a number of characters relative to *inFirstChar*.

Examples

```
// Using negative number for inFirstChar
$sub := Delete string("foobar.a4d"; -4; MAXTEXTLEN)
// $sub = "foobar"

// Using negative number for inNumChars
// -2 means to delete through the 2nd to last char
$sub := Delete string("12345"; 2; -2)
// $sub = "15"
```

enclose

version 2

enclose(inValue {; inEnclosures}) → Text

Parameter	Type	Description
inValue	<any>	→ Value to enclose
inEnclosures	Text	→ Characters to use for enclosure
Result	Text	← Enclosed text

Discussion

This command is a convenience for the common case when a value (usually a string) needs to be enclosed in some characters, such as double quotes.

The command first attempts to convert *inValue* to text according to the default formatting for its type. If *inValue* is not of a type that can be converted to text, an error is generated and execution is aborted.

If *inEnclosures* is not specified, the opening and closing enclosure characters default to double quotes (**Char(34)**).

If *inEnclosures* is specified and is a single character, the opening enclosure character is set to that character. The closing enclosure character is set according to the following table:

Opening enclosure	Closing enclosure
()
[]
{	}
<	>
Any other character	opening enclosure

If *inEnclosures* is specified and is more than one character, the opening enclosure character is set to the first character and the closing enclosure character is set to the second character.

Some examples:

Command	Output
<code>enclose("test")</code>	<code>"test"</code>
<code>enclose(13)</code>	<code>"13"</code>
<code>enclose("test"; "(")</code>	<code>(test)</code>
<code>enclose("test"; "*")</code>	<code>*test*</code>
<code>enclose("test"; "&")</code>	<code>&test;</code>

first not of

version 2
modified v5

`first not of(inSource; inMatchChars {; *})` → Longint

Parameter	Type	Description
<code>inSource</code>	Text	→ Text to search
<code>inMatchChars</code>	Text	→ Characters to search for
<code>*</code>	*	→ Perform a bitwise search
<code>Result</code>	Longint	← Position of first matching character

Discussion

This command searches *inSource* for the first character that does not match any of the characters in *inMatchChars* and returns its position. If no match is found, zero is returned.

If `*` is not passed, the search is case and diacritical-insensitive (the backwards-compatible default). However, unlike the **Position** command, composed characters like “ß” will not be matched with non-composed equivalents like “ss”.

If `*` is passed, the comparison is a strict bitwise comparison, which means case and diacriticals are significant.

first of**version 2
modified v5**

first of(inSource; inMatchChars {; *}) → Longint

Parameter	Type	Description
inSource	Text	→ Text to search
inMatchChars	Text	→ Characters to search for
*	*	→ Perform a bitwise search
Result	Longint	← Position of first matching character

Discussion

This command searches *inSource* for the first character that matches any of the characters in *inMatchChars* and returns its position. If no match is found, zero is returned.

If * is not passed, the search is case and diacritical-insensitive (the backwards-compatible default). However, unlike the **Position** command, composed characters like “ß” will not be matched with non-composed equivalents like “ss”.

If * is passed, the comparison is a strict bitwise comparison, which means case and diacriticals are significant.

format string**v5**

format string(inFormat; inArg0 {; ... argN}) → Text

Parameter	Type	Description
inFormat	Text	→ Format string
inArg0..inArgN	<any>	→ Format arguments
Result	Text	← Formatted text

Discussion

This command is the equivalent of:

```
inFormat %% (inArg0 {; inArg1 {; inArgN}})
```

For more information, see “%% (formatting operator)” on page 364.

identical strings

version 2

identical strings(inString1; inString2) → Boolean

Parameter	Type	Description
inString1	Text	→ Text to compare
inString2	Text	→ Text to compare
Result	Boolean	← True if completely identical

Discussion

This command does a strict case- and diacritical-sensitive comparison of the two strings and returns *True* if they are identical. This is very useful when comparing passwords.

Insert string

v5 (enhanced 4D)

Insert string(inSource; inWhat; inWhere) → Text

Parameter	Type	Description
inSource	Text	→ String in which to insert
inWhat	Text	→ String to insert
inWhere	Number	→ Position at which to insert
Result	Text	← New string

Discussion

The Active4D version of the **Insert string** command differs from 4D's version in that you can pass a negative number for *inWhere*, which is converted to the index of the Nth Unicode code unit from the end of *inSource*.

For example, to insert 4 characters from the end of a string, you could use:

```
$s := insert string("foobar.a4d"; ".0"; -4)
// $s = "foobar.0.a4d"
```

interpolate string

version 4.0

interpolate string(inString) → Text

Parameter	Type	Description
inString	Text	→ Text to interpolate
Result	Text	← Interpolated text

Discussion

This command interpolates *inString* according the rules for string interpolation as documented in “String Interpolation” on page 84.

Ordinarily you don’t need to use this command, because you can implicitly interpolate strings by using string literals with single-quotes. However, if you are dynamically building a string without string literals which you want to interpolate, you can use this command to do so.

last not of

version 2
modified v5

last not of(inSource; inMatchChars {; *}) → Longint

Parameter	Type	Description
inSource	Text	→ Text to search
inMatchChars	Text	→ Characters to search for
*	*	→ Perform a bitwise search
Result	Longint	← Position of first matching character

Discussion

This command searches *inSource* for the last character that does not match any of the characters in *inMatchChars* and returns its position. If no match is found, zero is returned.

If *** is not passed, the search is case and diacritical-insensitive (the backwards-compatible default). However, unlike the **Position** command, composed characters like “ß” will not be matched with non-composed equivalents like “ss”.

If *** is passed, the comparison is a strict bitwise comparison, which means case and diacriticals are significant.

last of**version 2
modified v5**

last of(inSource; inMatchChars {; *}) → Longint

Parameter	Type	Description
inSource	Text	→ Text to search
inMatchChars	Text	→ Characters to search for
*	*	→ Perform a bitwise search
Result	Longint	← Position of first matching character

Discussion

This command searches *inSource* for the last character that matches any of the characters in *inMatchChars* and returns its position. If no match is found, zero is returned.

If * is not passed, the search is case and diacritical-insensitive (the backwards-compatible default). However, unlike the **Position** command, composed characters like “ß” will not be matched with non-composed equivalents like “ss”.

If * is passed, the comparison is a strict bitwise comparison, which means case and diacriticals are significant.

left trim**version 2**

left trim(inString) → Text

Parameter	Type	Description
inString	Text	→ Text to trim
Result	Text	← Trimmed text

Discussion

This command trims control characters and whitespace from the beginning of *inString* and returns the result.

html encode

v5

html encode(inString {; inMode}) → Text

Parameter	Type	Description
inString	Text	→ Text to convert
inMode	Longint	→ How to convert
Result	Text	← Converted text

Discussion

This command encodes HTML special characters (like "&") to ISO HTML character entities (like "&"), according to *inMode* and the current output charset.

If *inMode* is not passed in, the conversion mode defaults to *A4D Encoding HTML*.

If *inMode* is an asterisk, the conversion mode is *A4D Encoding All*.

Otherwise you may pass a set of bit flags as you would to the **set output encoding** command. For more information on output encoding, see "set output encoding" on page 305.

Note: **html encode** will only convert non-ASCII characters if the output charset is ISO-8859-1 or ISO-8859-15.

mac to html

version 1
deprecated v5

mac to html(inString {; inMode}) → Text

Parameter	Type	Description
inString	Text	→ Text to convert
inMode	Longint	→ How to convert
Result	Text	← Converted text

Discussion

This command has been deprecated in favor of the renamed command **html encode**. It has been kept for backwards compatibility, but will be removed in a future version.

mac to utf8**version 4.5
deprecated v5**

mac to utf8(inText) → Text

Parameter	Type	Description
inString	Text	→ Text
Result	Text	← Original text

Discussion

As of v5, this command is deprecated and does nothing, since all text within Active4D is Unicode. It has been kept for backwards compatibility, but will be removed in a future version.

param text**version 2**

param text({*; inDelimiters} inSource; inReplaceParam {; ...inReplaceParamN}) → Text

Parameter	Type	Description
*	*	→ Indicates delimiters passed
inDelimiters	Text	→ Delimiters for parsing inSource
inSource	Text	→ Text to parse
inReplaceParam	<any>	→ Replacement parameter
Result	Text	← Transmogrified text

Discussion

This little beauty is in actuality a powerful little string processor with an embedded language.

At the simplest level, **param text** can be used to do multiple string replacements in one shot. All parameter placeholders indicated by “^N” in the source text are replaced with the corresponding replacement parameters, where *N* is the relative number of the replacement parameter. Replacement parameters are automatically converted to text.

For example:

```
// Current record of [Contacts] has Firstname="Buffalo",
// Lastname="Bill"

write(param text("Hello, ^1 ^2! How are you?"; \\
                [Contacts]Firstname; \\
                [Contacts]Lastname))

// output
Hello Buffalo Bill! How are you?
```

In addition to simple parameter replacement, you can specify alternate subsections of the source string to be used depending on the value of the replacement parameters.

Alternate subsections are marked in the form:

```
<delim>N<delim>T1<delim>T2<delim>
```

where *<delim>* is a delimiter character, *N* is a parameter number (one-based), *T1* is the first choice, and *T2* is the second choice. The delimiter characters are taken from the first two characters of *inDelimiters*. The default delimiters string is "#|", which is used if *inDelimiters* is less than two characters.

Three passes are made over the source string.

- First the delimiter *inDelimiters[[1]]* is used, and if the value of parameter *N* evaluates to zero, *T1* is used, else *T2* is used. A parameter is considered zero if it contains only numeric characters and converts to the number zero, or if its length = 0.
- The second pass uses the delimiter *inDelimiters[[2]]*, and if the value of parameter *N* evaluates to 1, *T1* is used, else *T2* is used.
- Finally, any occurrence of *^N*, where *N* is a replacement parameter number, is replaced with the corresponding replacement parameter.

Subsections with the same delimiter may not be nested.

The point of all this is to allow you to encode linguistic rules in your strings, rather than in your code. For example, here is a common problem:

```
$recs := records in selection([contact])

case of
  :($recs = 0)
    $msg := "There are no contacts available."
  :($recs = 1)
    $msg := "There is 1 contact available."
  else
    $msg:= "There are " + string($recs) + "contacts available"
end case
```

Multiply this by a hundred messages, and you can see why no one bothers to be grammatically correct with such messages.

Using **param text**, you can easily handle such a situation like this:

```
$recs := records in selection([contacts])
$msg := "There |1|is|are| #1#no#^1# ^2|1||s| available."
$msg := param text($msg; "#|"; $recs; "contact")
// This code does the same as the case statement above.
```

Okay, now we need to decode the source string.

Element	Description
There	literal text
	start a singular/non-singular selector
1	use replacement parameter #1, \$recs, as the selector
is are	insert "is" if \$recs evaluates to 1, "are" otherwise
#	start a zero/non-zero selector
1	use replacement parameter #1, \$recs, as the selector
#no#^1#	insert "no" if \$recs evaluates to zero, replacement parameter #1 (\$recs) if not
^2	insert replacement parameter #2, "contact"
	start a singular/non-singular selector
1	use replacement parameter #1, \$recs, as the selector
s	do nothing if \$recs evaluates to 1, insert "s" otherwise

It takes some getting used to, but once you master this mini-language you can eliminate a lot of code by programming your strings!

Note: One useful technique is to use a boolean flag as a selector between alternate text. Simply pass a replacement parameter of **String(Num(\$myBool))** and use a zero/non-zero selector.

Position

**version 2 (enhanced 4D)
modified v5**

Position(inFind; inSource {; inStart {; { *; } { outLengthFound {; * }}}} → Longint

Parameter	Type	Description
inFind	Text	→ Text to search for
inSource	Text	→ Text in which to search
inStart	Longint	→ Starting position for search
*	*	→ Search in reverse
outLengthFound	Longint	→ Length of matched text
*	*	→ Use bitwise comparison
Result	Longint	← Position of first match

Discussion

Active4D extends this 4D command by allowing you to do a reverse search.

If *inStart* is not passed, it defaults to 1. If *inStart* is passed and is positive, the search starts at *inSource*[[*inStart*]]. If *inStart* is passed and is negative, the search starts *inStart*

characters from the end of *inSource*, so *inStart* of -1 would mean to start searching from the last character of *inSource*.

If a reverse search is performed, *inStart* defaults to the end of *inSource* if it is not passed. Reverse searches begin at *inSource*[[*inStart*]] and proceed towards *inSource*[[1]]. Note that when searching in reverse, the start position marks the *end* of the portion of *inSource* that is searched. In other words, it is as if you are looking for the last occurrence of *inFind* within **Substring**(*inSource*; 1; *inStart*).

outLengthFound and the final * have the same function as the last two parameters of the **Position** command in 4D.

Example

```
$source := "test123test"
$pos := position("test"; $source)
// forward search, $pos = 1

$pos := position("test"; $source; *)
// reverse search from end, $pos = 8

$pos := position("test"; $source; 2)
// forward search starting at 2nd char, $pos = 8

$pos := position("test"; $source; 7; *)
// reverse search starting at 7th char, $pos = 1

$pos := position("test"; $source; -2)
// forward search starting at 2nd char from end, $pos = 0

$pos := position("test"; $source; -1; *)
// reverse search starting at 1st char from end, $pos = 8

$pos := position("test"; $source; -5; *)
// reverse search starting at 5th char from end, $pos = 1
```

right trim

version 2

right trim(*inString*) → Text

Parameter	Type	Description
<i>inString</i>	Text	→ Text to trim
Result	Text	← Trimmed text

Discussion

This command trims control characters and whitespace from the end of *inString* and returns the result.

slice string**version 4.0
modified v5**

```
slice string(inString; inDelimiter {; *}) → Text
```

Parameter	Type	Description
<i>inString</i>	Text	→ Text to slice
<i>inDelimiter</i>	Text	→ Where to slice
<i>outRemainder</i>	Text	← Text after delimiter
*	*	→ Bitwise matching
Result	Text	← Text before delimiter

Discussion

This command slices a string into two parts that are separated by delimiter. If *inDelimiter* is found in *inString*, the text before the delimiter is returned. If *outRemainder* is passed, it is set to the text after the delimiter.

If * is passed, delimiter matching is bitwise (case and diacritical sensitive). Otherwise the default (backward-compatible) behavior is to ignore case and diacriticals, and composed characters will be matched. For example, if the delimiter is “ae”, the composed character “æ” will be matched.

If *inDelimiter* is not found, *inString* is returned whole and *outRemainder* is set to an empty string if it is passed.

split string**version 2
modified v5**

```
split string(inSource; inPattern; outArray {inPatternsChars
  {; inCaseSensitive {; inLimit}}}) → Longint
```

Parameter	Type	Description
inSource	Text	→ Text to split
inPattern	Text	→ Provides splitting boundaries
outArray	String/Text Array	→ Receives chunks of inSource
inPatternsChars	Boolean	→ True if inPattern should be treated as an array of characters to match
inCaseSensitive	Boolean	→ How to match inPattern
inLimit	Number	→ Total number of splits that may be created
Result	Longint	← Number of splits made

Discussion

This command splits *inSource* into chunks and puts the chunks into *outArray*, according to the string *inPattern*.

If *inPatternsChars* is *True* (the default if not passed), the command splits the source string at any of the characters in *inPattern*. Pattern characters at the beginning and end of the source are stripped out, and empty elements are ignored.

If *inPatternsChars* is *False*, the command splits the source string at any occurrence of the entire pattern. Empty elements anywhere in the source are maintained.

Case-sensitivity of pattern matching is set by *inCaseSensitive*. If not passed, matching is case sensitive by default. If *inPatternsChars* is *False* and *inCaseSensitive* is *False*, **split string** will match composed characters. For example, if the pattern is "ae", the composed character "æ" will be matched.

inLimit limits the number of chunks that are created. Once the limit is reached, one more chunk is created with the rest of the string, whether or not it contains the pattern.

This command is especially useful for splitting formatted text such as tab-delimited strings.

```
$source := "This is a test"
split string($source; " "; $chunks)
// $chunks now contains 4 elements: "This", "is", "a", "test"

split string($source; " "; $chunks; true; true; 2)
// $chunks now contains 2 elements: "This", "is a test"

split string("1||3|"; "|"; $params)
// $params contains 2 elements: "1", "3"

split string("1||3|"; "|"; $params)
// $params contains 4 elements: "1", "", "3", ""
```

String

version 1 (enhanced 4D)
modified v5

String(inValue {; inFormat}) → Text

Parameter	Type	Description
inValue	<any>	→ Value to write to the response buffer
inFormat	Text Number	→ Format to use when converting to text
Result	Text	← Converted value

Discussion

The Active4D version of the **String** command differs from 4D's version in that:

- If you pass a *Boolean* value with a format, it will be converted to 1 or 0, which allows you to use formats like "Yes;;No".
- If a BLOB is passed in, it is assumed to be text. You can specify the text format within the BLOB by passing the relevant constant (such as *UTF8 Text without length*) in the *inFormat* parameter. If no format is passed, the text format is assumed to be *UTF8 Text without length*.

Substring

version 1 (enhanced 4D)

Substring(inString; inFirstChar {; inNumChars}) → Text

Parameter	Type	Description
inString	Text	→ String from which to get substring
inFirstChar	Number	→ Position of first character of substring
inNumChars	Number	→ Size of substring
Result	Text	← Substring

Discussion

The Active4D version of the **Substring** command differs from 4D's version in that you can pass a negative number for *inFirstChar* and *inNumChars*, which is converted to the index of the *Nth* Unicode code unit from the end of *inString*.

When a negative number is passed for *inNumChars*, note that it signifies an index relative to the end of the string, not a number of characters relative to *inFirstChar*.

Examples

```
// Using negative number for inFirstChar
$sub := substring("foobar.a4d"; -3)
// $sub = "a4d"

// Using negative number for inNumChars
// -2 means to extract through the 2nd to last char
$sub := substring("12345"; 2; -2)
// $sub = "234"
```

trim

version 2

trim(inString) → Text

Parameter	Type	Description
inString	Text	→ Text to trim
Result	Text	← Trimmed text

Discussion

This command trims control characters and whitespace from the beginning and end of *inString* and returns the result.

url decode

version 2

url decode(inString) → Text

Parameter	Type	Description
inString	Text	→ Text to decode
Result	Text	← Decoded text

Discussion

This command is a synonym for **url decode query**.

url decode path

version 2

url decode path(inString) → Text

Parameter	Type	Description
inString	Text	→ Text to decode
Result	Text	← Decoded text

Discussion

This command decodes a URL-encoded string as a URL-encoded path. For more information on URL encoding and decoding, see “URL Encoding/Decoding” on page 361.

url decode query

version 2

url decode query(inString) → Text

Parameter	Type	Description
inString	Text	→ Text to decode
Result	Text	← Decoded text

Discussion

This command decodes a URL-encoded string as a URL-encoded query string. For more information on URL encoding and decoding, see “URL Encoding/Decoding” on page 361.

url encode

version 2

url encode(inString) → Text

Parameter	Type	Description
inString	Text	→ Text to encode
Result	Text	← Encoded text

Discussion

This command is a synonym for **url encode query**.

url encode path

version 2

url encode path(inString) → Text

Parameter	Type	Description
inString	Text	→ Text to encode
Result	Text	← Encoded text

Discussion

This command encodes *inString* into a URL-encoded path. For more information on URL encoding and decoding, see “URL Encoding/Decoding” on page 361.

url encode query

version 2

url encode query(inString) → Text

Parameter	Type	Description
inString	Text	→ Text to encode
Result	Text	← Encoded text

Discussion

This command encodes *inString* into a URL-encoded query parameter name or value. For more information on URL encoding and decoding, see “URL Encoding/Decoding” on page 361.

utf8 to mac**version 4.5
deprecated v5**

utf8 to mac(inString) → Text

Parameter	Type	Description
inString	Text	→ Text to convert
Result	Text	← Converted text

Discussion

This command is deprecated and does nothing, since all text is Unicode within Active4D. It has been kept for backward compatibility, but will be removed in a future version.

System Documents

Active4D implements the most important document commands from 4D. In addition, it adds many new commands for manipulating files and directories and working with URL-style paths.

Document Paths

As in 4D, a path passed to a document command may be *full* or *partial*, also known as *absolute* and *relative*. In addition, the path may be *native* or *URL-style*.

If a path is native, it is considered a full path if it begins with a drive name and ends with either a directory name (for commands that take a directory) or a filename (for commands that take a filename). If the path is URL-style, it is considered a full path if it begins with '/'.

If a path is native, it is considered a partial path if it does not begin with a drive name. If a path is URL-style, it is considered a partial path if it does not begin with '/'.

A partial path is relative to the *default directory* (see “The Default Directory” on page 42).

The *last* element of a full or partial path, whether a directory or filename, may be an alias (also known as a “shortcut” on Windows). Directories *within* a path may not be aliases.

Document Command Enhancements

There are several important enhancements to document commands that Active4D implements:

- For security reasons, all paths are checked to ensure they are either within the web root directory or within one of the directories specified by the “safe doc dirs” option in Active4D.ini. Aliases are resolved prior to checking.

If you want to use document commands on files outside of the root directory, you must add the directory path to the “safe doc dirs” path list in Active4D.ini. For information on Active4D.ini, see Chapter 3, “Configuration.”
- You may pass a URL-style path (using '/' as the directory separator) instead of a native path. If no '/' is in a path, it is assumed to be a native path.
- To ensure proper functioning of your scripts, Active4D closes all documents that have been opened with document commands when the script finishes executing. This prevents the situation where a script error leaves a file open and thus prevents other scripts from writing to the file.

Affected Commands

The document commands affected by the enhancements mentioned above are:

The file upload command **copy upload** is also affected by these enhancements. See “copy upload” on page 196 for more information.

Append document	FOLDER LIST
COPY DOCUMENT	MOVE DOCUMENT
Create document	Open document
DELETE DOCUMENT	READ PICTURE FILE
DELETE FOLDER	Test path name
DOCUMENT LIST	WRITE PICTURE FILE

Error Codes

All document-related commands set the *Error* variable to zero on success and an error code on failure. *OK* is set for commands that perform an operation and do not return an error code.

If a document command returns an error number -70001 or lower, it is a standard C library error code and not a Macintosh error code. The actual C error code is:

```
abs(error) - 70000
```

So, for example, -70001 is 1, -70002 is 2, etc. C library error codes can be found here:

<http://www.virtsync.com/c-error-codes-include-errno>

Working With Large Files

Active4D's document commands can work with files larger than 2GB. However, since text and BLOBs are limited to 2GB in length, you cannot read or write more than 2GB at a time.

Append document

(modified 4D) version 2
modified v5

Append document(*inPath* {; *inType*}) → DocRef

Discussion

The Active4D version of this command differs from the 4D version in that:

- Passing an empty string for *inPath* will do nothing other than set *OK* to zero.
- *inType* is ignored. The filename extension must be part of *inPath*.

Warning: Active4D maintains its own list of open documents, and the document references returned by this command in Active4D are **not** interchangeable with the document references returned by document commands within 4D methods.

Create document

(modified 4D) version 2
modified v5

Create document(*inPath* {; *inType*}) → DocRef

Discussion

The Active4D version of this command differs from the 4D version in that:

- Passing an empty string for *inPath* will do nothing other than set *OK* to zero.
- *inType* is ignored. The filename extension must be part of *inPath*.

Warning: Active4D maintains its own list of open documents, and the document references returned by this command in Active4D are **not** interchangeable with the document references returned by document commands within 4D methods.

current file

version 2

current file → Text

Parameter	Type	Description
Result	Text	← Filename of currently executing script

Discussion

This command returns the filename of the currently executing script. It is exactly equivalent to **filename of(current path)**.

current path

version 2
modified version 4.0

current path {{{inWantLibraryPath}{*}}} → Text

Parameter	Type	Description
inWantLibraryPath	Boolean	→ Return path to library
*	*	→ Return web root-relative path
Result	Text	← Path of currently executing script

Discussion

This command returns the path of the currently executing script in URL format. Unlike **requested url**, if this command is executed within an include file, the path of the include file is returned.

If executed within a library, the path to the library is returned, unless *inWantLibraryPath* is passed and is *False*, in which case the path to the nearest script file in the source stack (if any) is returned.

Note: If this command is executed within a library and *inWantLibraryPath* is not passed or is passed and is *True*, the * option is not supported.

If executed within a text block, the path to the nearest script file in the source stack (if any) is returned.

If the * is passed, the path returned is an absolute web root-relative path.

Example

Let us assume we are using the default web root, "web", whose path is the following (as returned by **get root**):

```
/HD/WebApps/MyApp/web/
```

If a script called “act_mailReminder.a4d” within the “login” directory within the web root executes **current path**. The result will be:

```
/HD/WebApps/MyApp/web/login/act_mailReminder.a4d
```

If the same script executes **current path(*)**, the result will be:

```
/login/act_mailReminder.a4d
```

default directory

version 2

default directory{(*)} → Text

Parameter	Type	Description
*	*	→ Use URL format for path
Result	Text	← Path to “default” directory

Discussion

This command returns a full path to the “default” directory. For information on the default directory, see “The Default Directory” on page 42.

If the * is not passed, a native path is returned with a native directory separator at the end. Passing * returns the path in URL format with ‘/’ at the end.

DELETE FOLDER

version 2

DELETE FOLDER(inPath {; *})

Parameter	Type	Description
inPath	Text	→ Path of folder to delete
*	*	→ Recursively delete contents

Discussion

If * is not passed, this command works exactly as its 4D counterpart does, in that only empty directories may be deleted. If * is passed, this command recursively deletes the directory and all of its contents.

Warning: Using the DELETE FOLDER command in recursive mode is of course extremely dangerous. Be absolutely sure you know what you are deleting before using the recursive mode of this command.

directory exists

version 4.0

directory exists(inPath) → Boolean

Parameter	Type	Description
inPath	Text	→ URL or native path
Result	Boolean	← True if is an existing directory

Discussion

This command is the equivalent of:

```
test path name(inPath) = Is a directory
```

In other words, it will return *True* if the entity described by *inPath* exists and is a directory.

directory of

version 2

directory of(inPath {; *}) → Text

Parameter	Type	Description
inPath	Text	→ URL or native path
*	*	→ Suppress trailing separator
Result	Text	← Directory portion of given path

Discussion

This command returns the directory portion of the given path with a trailing directory separator, unless *** is passed, in which case the trailing separator is suppressed.

If *inPath* contains a forward slash (*/*), it is assumed to be a URL-style path.

directory separator

version 2

directory separator → Text

Parameter	Type	Description
Result	Text	← Native directory separator

Discussion

This command returns the native directory separator for the currently running platform, which is *:"* on Macintosh and *\"* on Windows.

extension of**version 2**extension of(*inPath*) → Text

Parameter	Type	Description
<i>inPath</i>	Text	→ URL or native path
Result	Text	← Filename extension of given path

Discussion

This command returns the filename extension, including the dot, of the given path. If the path ends in a directory, an empty string is returned.

file exists**version 4.0**file exists(*inPath*) → Boolean

Parameter	Type	Description
<i>inPath</i>	Text	→ URL or native path
Result	Boolean	← True if is an existing file

Discussion

This command is the equivalent of:

```
test path name(inPath) = Is a document
```

In other words, it will return *True* if the entity described by *inPath* exists and is a file.

filename of**version 2
modified v5**filename of(*inPath* {; *}) → Text

Parameter	Type	Description
<i>inPath</i>	Text	→ URL or native path
*	*	→ Strip extension
Result	Text	← Filename portion of given path

Discussion

This command returns the filename portion of the given path. If *inPath* contains a forward slash ('/'), it is assumed to be a URL-style path.

If `*` is passed, the returned filename is stripped of its extension.

get root

version 1
modified v5

get root(*inVirtualHost*) → Text

Parameter	Type	Description
<i>inVirtualHost</i>	Text	→ Virtual host name
Result	Text	← Full path of web root

Discussion

This command returns a full URL-style path to the web root directory, with a trailing `'/'`.

If you have defined virtual hosts in `VirtualHosts.ini` and *inVirtualHost* is not passed, **get root** will return the mapped web root of the current virtual host.

If *inVirtualHost* is passed and such a named virtual host exists in the virtual host table, the web root directory of that virtual host is returned. If no such named virtual host exists, the current web root directory is returned.

This command is especially useful when building a path for use with the document commands, as in:

```
$path := join paths(get root; "templates/invoice.txt")
$docRef := open document($path)
```

join paths

version 4.0

join paths(***; *inPathSegment1* { ...*inPathSegmentN*}) → Text

Parameter	Type	Description
<i>*</i>	<i>*</i>	Force result to native path
<i>inPathSegmentN</i>	Text	→ Path to convert
Result	Text	← Joined path

Discussion

This command concatenates one or more path segments together intelligently. If any path segment is an absolute path, all previous segments are thrown away, and joining continues. The return value is the concatenation of the segments, with exactly one directory separator inserted between segments.

If the optional `*` is not passed, the resulting path is in URL (Unix) format. If it is passed, the resulting path is in native format.

This command is designed to relieve of the burden of checking whether or not directory path segments have a trailing directory separator, as in this example:

```
$docName := "2005-08-27.txt"
open document(join paths(get root; "news"; $docName))

// result of join paths:
/HD/Users/Homer/Documents/site/web/news/2005-08-27.txt
```

MOVE DOCUMENT

(modified 4D) version 2
modified v5

MOVE DOCUMENT(inSourcePath; inDestPath {; *})

Parameter	Type	Description
inSourcePath	Text	→ Path to existing document/folder
inDestPath	Text	→ Destination path
*	*	→ Force move

Discussion

The Active4D version of this command differs from the 4D version in that:

- If the optional * parameter is passed, the move is forced, even if a document exists at the destination path.
- Folders may be renamed in addition to documents.

native to url path

version 2

native to url path(inPath) → Text

Parameter	Type	Description
inPath	Text	→ Native path to convert
Result	Text	← Full path of web root

Discussion

This command converts a native path for URL-style path to a native path for the currently running platform.

Open document

(modified 4D) version 2
modified v6.3r1

Open document(*inPath* {; *inType*} {; *inMode*}) → DocRef

Discussion

The Active4D version of this command differs from the 4D version in that:

- Passing an empty string for *inPath* will do nothing other than set *OK* to zero.
- *inType* is ignored. The filename extension must be part of *inPath*.

As of v5, the **Open document** command now conforms more closely to 4D's behavior. In particular, an exclusive write lock is acquired on files opened in read/write or write mode. If no open mode is passed and a file is already open for writing, the file will be opened in read-only mode.

As of v6.2r1, *inType* may be omitted as in 4D.

Warning: Active4D maintains its own list of open documents, and the document references returned by this command in Active4D are **not** interchangeable with the document references returned by document commands within 4D methods.

RECEIVE PACKET

(modified 4D) version 2

RECEIVE PACKET(*inDocRef*; *inPacket*; *inStopChar* | *inNumChars*)

Discussion

This command differs from the 4D version in that it may only be used with documents, thus the document reference is required.

requested url

version 2

requested url{*} → Text

Parameter	Type	Description
*	*	→ Pass to return physical path
Result	Text	← Virtual or physical path

Discussion

This command returns the path to the script that was requested in the current execution. If * is not passed, the root-relative virtual path (including /4DCGI) is returned. If * is passed, the full physical path to the requested script is returned in URL format.

```
writebr(requested url)    // virtual
writebr(requested url(*)) // physical

// output using 4D as network layer
/4DCGI/test.a4d
/Supreme/Users/aparajit/Development/Active4D/web/test.a4d
```

resolve path

v5

resolve path(inPath {; *}) → Text

Parameter	Type	Description
inPath	Text	→ Path to resolve
*	*	→ Return native path
Result	Text	← Absolute path

Discussion

This command resolves relative paths, directory movement and aliases or shortcuts in *inPath* and returns an absolute full path. The path does not actually have to exist to be resolved.

If * is not passed, the returned path is in URL (Posix) format. If * is passed, the returned path is in native 4D format (HFS on macOS).

If an error occurs while resolving the path, *Error* will contain an appropriate error code.

SEND PACKET

(modified 4D) version 2

SEND PACKET(inDocRef; inPacket)

Discussion

This command differs from the 4D version in that it may only be used with documents, thus the document reference is required.

SET DOCUMENT POSITION

(modified 4D) version 2

SET DOCUMENT POSITION(inDocRef; inOffset {; inAnchor})

Discussion

Named constants have been provided for the *inAnchor*: *Position from start*, *Position from end*, and *Position from current*.

split path

v5

split path(inPath; outDirectory; outFilename {; *}) → Text

Parameter	Type		Description
inPath	Text	→	Path to split
outDirectory	Text	←	Directory portion of inPath
outFilename	Text	←	Filename portion of inPath
*	*	→	Suppress directory separator

Discussion

This command converts splits a path. *outDirectory* receives the directory portion of the path and *outFilename* receives the filename portion of the path.

If *inPath* has no directory separators, *outDirectory* will be empty and *outFilename* will contain *inPath*.

If *inPath* ends with a directory separator, *outFilename* will be empty.

If * is not passed and *inPath* has more than one path component, *outDirectory* will have a trailing directory separator. If * is passed, the trailing directory separator in *outDirectory* is suppressed.

url to native path

version 2

url to native path(inPath) → Text

Parameter	Type	Description
inPath	Text	→ URL-style path to convert
Result	Text	← Full path of web root

Discussion

This command converts a URL-style path to a native path for the currently running platform.

Timestamps

A common practice in database design is to define a *timestamp* field as part of each table. Timestamps are often used to mark creation date/time, modification date/time, etc.

Timestamp Format

A timestamp combines a date and time together into one Alpha field with a width of 17 characters. The format of a timestamp is as follows:

```
YYYYMMDDhhmmsssttt  
  
YYYY = 4-digit year  
MM = 2-digit month  
DD = 2-digit day  
hh = 2-digit hour (0-23)  
mm = 2-digit minute  
ss = 2-digit second  
ttt = 3-digit thousandths of a second (milliseconds)
```

By using this format, only one field can be used to store both the date and time. More importantly, a timestamp is formatted such that records can be sorted and compared by the timestamp, since the elements of the timestamp progress from most significant to least significant.

Note: It is up to you to properly define the timestamp fields in your database.

Timestamp Time

To ensure consistency in timestamps, timestamps dates and times are converted from the local time zone to Coordinated Universal Time, also known as UTC or GMT.

When working with timestamps, you always deal with the local time zone. All timestamp commands which create a timestamp expect a local date and time, and all timestamp commands that return a portion of the timestamp return a local date and/or time.

Timestamp Normalization

Timestamps are always kept in a *normalized* state — the date and time always contain valid numbers. If a timestamp is created with a date or time where an element of the date or time is beyond the valid range for that element, the extra is factored into the next most significant element. If that element is then beyond the valid range, the process continues until all elements are normalized.

For example, suppose you create a timestamp with the time 07:59:121. In this case, 121 seconds would be factored into the minutes, leaving us with two extra minutes and one second. The two extra minutes would then get factored into the hour, leaving us with one extra hour and one minute. So the normalized time would be 08:01:01.

Normalization allows you to do things like creating a timestamp which represents the 270th day of the year, like this:

```
$ts := timestamp(!01/270/01!)  
// $ts = "20010927050000000"
```

Using Timestamps with Optimistic Locking

In multiuser databases such as those served on the web, timestamps are especially valuable in solving the problem of record contention. For example, imagine this scenario:

- 1 User A begins editing a record.
- 2 User B begins editing the same record.
- 3 User B saves the changes to the record.
- 4 User A saves the changes to the record.

This is a situation you want to prevent, because User A's changes will destroy the changes User B made — changes which could very well be critical. In fact, once User B saves the changes, User A's copy of the data is essentially incorrect.

In a Client/Server database, you typically solve this problem by locking a record as soon as the user begins editing it. This is known as *pessimistic locking*, which leads to this scenario:

- 1 User A begins editing a record, locking that record to others.
- 2 User B attempts to edit the same record, is told to wait until User A is finished.
- 3 User A saves the changes and releases the lock.
- 4 User B is allowed to edit the record.
- 5 User B saves the changes to the record.

This technique is not without its problems. What if User A goes to lunch in the middle of editing the record? What if User B just wants to view the record without changing it? These problems can and have been solved in various ways, but the solutions are often cumbersome.

Another approach which solves these problems is called *optimistic locking*. Here's how it works:

- 1 User A begins editing a record. The modification timestamp of the record is saved in the user's session for later comparison.
- 2 User B begins editing the same record. The modification timestamp of the record is saved in the user's session for later comparison.
- 3 User B attempts to save the changes to the record. Since the timestamp of the record matches the saved timestamp, the save is allowed and the timestamp is updated to the modification time.
- 4 User A attempts saves the changes to the record. Because the saved timestamp does not match the record's actual timestamp, the save is not allowed and the user is asked to try editing the record again.

This methodology is known as *optimistic locking* because it assumes that in the majority of cases there will not be a conflict — which is indeed the case in the majority of applications.

If you are writing a multiuser web application, you should definitely consider using timestamps and optimistic locking.

timestamp**version 2**

timestamp → Text
 timestamp(inDate {; inTime}) → Text
 timestamp(inYear; inMonth; inDay;
 inHour; inMinute; inSecond; inMilliseconds) → Text

Parameter	Type		Description
inDate	Date	→	Local date
inTime	Time	→	Local time
inYear	Number	→	Local year
inMonth	Number	→	Local month
inDay	Number	→	Local day
inHour	Number	→	Local hour
inMinute	Number	→	Local minute
inSeconds	Number	→	Seconds
inMilliseconds	Number	→	Milliseconds
Result	Text	←	Timestamp

Discussion

This command has three forms. The first form has no parameters and returns a timestamp for the current date and time.

The second form takes a 4D date and time in the local time zone and creates a timestamp from those values. If *inTime* is not passed, it is assumed to be midnight in the local time zone (00:00:00).

The third form specifies the exact values for each element of the timestamp. The values are in the local time zone.

The result of each form is a 17-character string containing the given date and time, normalized to valid values and converted from the local time zone to UTC.

add to timestamp

version 2

add to timestamp(inTimestamp; inYears {; inMonths {; inDays {; inHours {; inMinutes {; inSeconds {; inMilliseconds}}}}}) → Text

Parameter	Type	Description
inYear	Number	→ Local year
inMonth	Number	→ Local month
inDay	Number	→ Local day
inHour	Number	→ Local hour
inMinute	Number	→ Local minute
inSeconds	Number	→ Seconds
inMilliseconds	Number	→ Milliseconds
Result	Text	← Timestamp

Discussion

This command adds the specified number of date or time elements to *inTimestamp* and returns the result. Any of the values passed in may be negative.

For example, to get a timestamp from 30 days ago, you would use this code:

```
$monthAgo := add to timestamp(timestamp; 0; 0; -30)
```

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

timestamp difference

version 4.0

timestamp difference(inTimestamp1; inTimestamp2) → Real

Parameter	Type	Description
inTimestamp1	Text	→ A timestamp
inTimestamp2	Text	→ A timestamp
Result	Real	← Difference in seconds

Discussion

This command subtracts *inTimestamp2* from *inTimestamp1* and returns the difference between them in seconds.

timestamp string

version 2

timestamp string(*inTimestamp*) → Text

Parameter	Type	Description
<i>inTimestamp</i>	Text	→ A timestamp
Result	Text	← Formatted timestamp

Discussion

This command formats *inTimestamp* as a local date and time in the form:

```
YYYY-MM-DD hh:mm:ss.ttt
```

To format a timestamp in another way, use the other timestamp commands to extract the timestamp elements and format them as necessary.

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

timestamp date

version 2

timestamp date(*inTimestamp*) → Date

Parameter	Type	Description
<i>inTimestamp</i>	Text	→ A timestamp
Result	Date	← Date of the timestamp

Discussion

This command returns the date of the given timestamp, converted from UTC to the local time zone.

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

timestamp time

version 2

timestamp time(inTimestamp) → Time

Parameter	Type	Description
inTimestamp	Text	→ A timestamp
Result	Time	← Time of the timestamp

Discussion

This command returns the time of the given timestamp, converted from UTC to the local time zone.

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

get timestamp datetime

version 2

get timestamp datetime(inTimestamp; outDate; outTime)

Parameter	Type	Description
inTimestamp	Text	→ A timestamp
outDate	Date variable	← Local date of timestamp
outTime	Time variable	← Local time of timestamp

Discussion

This command returns the date and time of the given timestamp, converted from UTC to the local time zone.

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

timestamp year

version 2

timestamp year(inTimestamp) → Number

Parameter	Type	Description
inTimestamp	Text	→ A timestamp
Result	Number	← Year of the timestamp

Discussion

This command returns the year of the given timestamp, converted from UTC to the local time zone.

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

timestamp month

version 2

timestamp month(inTimestamp) → Number

Parameter	Type	Description
inTimestamp	Text	→ A timestamp
Result	Number	← Month of the timestamp

Discussion

This command returns the month of the given timestamp, converted from UTC to the local time zone.

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

timestamp day

version 2

timestamp day(inTimestamp) → Number

Parameter	Type	Description
inTimestamp	Text	→ A timestamp
Result	Number	← Day of the timestamp

Discussion

This command returns the day of the month of the given timestamp, converted from UTC to the local time zone.

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

timestamp hour

version 2

timestamp hour(inTimestamp) → Number

Parameter	Type		Description
inTimestamp	Text	→	A timestamp
Result	Number	←	Hour of the timestamp

Discussion

This command returns the hour of the given timestamp, converted from UTC to the local time zone.

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

timestamp minute

version 2

timestamp minute(inTimestamp) → Number

Parameter	Type		Description
inTimestamp	Text	→	A timestamp
Result	Number	←	Minute of the timestamp

Discussion

This command returns the minute of the given timestamp, converted from UTC to the local time zone.

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

timestamp second

version 2

timestamp second(inTimestamp) → Number

Parameter	Type	Description
inTimestamp	Text	→ A timestamp
Result	Number	← Second of the timestamp

Discussion

This command returns the second of the given timestamp.

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

timestamp millisecond

version 2

timestamp millisecond(inTimestamp) → Number

Parameter	Type	Description
inTimestamp	Text	→ A timestamp
Result	Number	← Millisecond of the timestamp

Discussion

This command returns the millisecond of the given timestamp.

If *inTimestamp* is not 17 characters, all of which are digits, an error is generated and execution is aborted.

User Authentication

User authentication requires a coordinated effort between the `Realms.ini` config file (which defines your security realms), the `On Authenticate` event handler (which must be defined to handle authentication), and the commands in this section.

For information on `Realms.ini`, see “`Realms.ini`” on page 47. For information on the `On Authenticate` event handler, see “`On Authenticate`” on page 129.

auth password

version 2

auth password → Text

Parameter	Type	Description
Result	Text	← Password of authenticated user

Discussion

This command returns the password of the authenticated user for the current realm, if any.

auth type

version 2

auth type → Text

Parameter	Type	Description
Result	Text	← Type of authentication used

Discussion

This command returns the type of authentication used. Currently the only authentication type supported is "Basic".

auth user

version 2

auth user → Text

Parameter	Type	Description
Result	Text	← Username of authenticated user

Discussion

This command returns the username of the authenticated user for the current realm, if any. If **current realm** returns a non-empty string and **auth user** returns an empty string, you know that the user has not yet been authenticated.

authenticate**version 2**

authenticate{(inRealm)}

Parameter	Type	Description
inRealm	Text	→ Realm to pass to browser

Discussion

This command generates the proper response status and headers to prompt a user authentication dialog on the client browser. If *inRealm* is not passed, the current realm as defined in Realms.ini is automatically passed to the client browser, otherwise *inRealm* is used.

Browsers cache the username and password for a given realm until the browser is closed. This is a potential security risk if the user leaves a publicly accessible browser open. In addition, if the user logs out or is inactive for a long period without closing the browser, you may decide that the user must be reauthenticated.

Some old browsers (most notably IE4) may not display an authentication dialog if passed a realm that has already been authenticated. You can force the browser to authenticate a user by changing the realm to some unique name, for example by appending the **Tickcount**. After the user enters his or her credentials, *current realm* will still return the original unadulterated realm name, since it is looked up in Realms.ini.

current realm**version 2**

current realm → Text

Parameter	Type	Description
Result	Text	← Name of current realm

Discussion

This command returns the name of the current realm, if any. The realm is determined by the Realms.ini config file.

Variables

Active4D provides a number of commands that extend what you can do with and to local variables in your scripts.

Many of these commands were created for use with Active4D's debugging tools.

defined**version 1 (modified v2)**

```
defined(inVariable | *; inVarName) → Boolean
```

Parameter	Type	Description
inVariable	Variable	→ Actual variable to test
*	*	→ Check by name
inVarName	Text	→ Name of variable to test
Result	Boolean	← True if defined in current scope

Discussion

This command tests the existence of a variable. You may either pass a variable reference directly, or you may pass * followed by a variable name. Local variable names being tested should begin with '\$'.

This command tests whether or not a local variable has been assigned a value yet, and thus has been defined.

get local**version 2**

```
get local(inName) → <any>
```

Parameter	Type	Description
inName	Text	→ Name of local variable to lookup
Result	<any>	← Value of given local variable

Discussion

Given a local variable name (including the leading '\$'), this command returns the variable's value if it is both defined in the current scope and not an array, or an empty string if not.

local variables

version 2
modified version 3

local variables{(inScope)} → Iterator

Parameter	Type	Description
inScope	Text	→ Name of library.method for which to retrieve local variables
Result	Iterator	← Iterator to local variable collection

Discussion

This command returns an iterator which you can use to iterate over the local variables in either the current scope, if *inScope* is not passed, or in a named scope, if *inScope* is passed.

If *inScope* is passed, it must be in the form *library.method*, except for the top level scope, which must be called "main".

The keys of the collection referenced by the iterator are the variables names and the values are the variable values, which can be of any type.

set local

version 2
modified v6.0

set local(inName; inValue {; *})

Parameter	Type	Description
inName	Text	→ Name of variable to set
inValue	<any>	→ Value to set for the given variable
*	*	→ If passed, arrays are copied

Discussion

This command searches the local variables in the current scope for the variable with the name *inName*, which should include the leading '\$'.

If the variable is found, the variable's value is replaced with *inValue*. Otherwise a new local variable is created with the given name.

If the value is an array and * is not passed, the current value of the array is used as the value. If * is passed, the entire array is copied.

type descriptor

version 2

type descriptor(inVariable) → Text

Parameter	Type	Description
inVariable	Variable	→ Actual variable reference
Result	Text	← Textual description of variable's type

Discussion

This command returns a textual description of the variable's type, exactly as it would appear in the 4D debugger.

undefined

version 2

undefined(inVariable | *; inVarName) → Boolean

Parameter	Type	Description
inVariable	Variable	→ Actual variable to test
*	*	→ Check by name
inVarName	Text	→ Name of variable to test
Result	Boolean	← True if undefined in current scope

Discussion

This command is exactly equivalent to **not(defined(inVariable))**.

variable name

version 2

variable name(*inVariable*) → Text

Parameter	Type	Description
<i>inVariable</i>	Variable	→ Actual variable to get name of
Result	Text	← Name of the variable

Discussion

Given an actual variable reference, this command returns the name of variable, or the name of the referent if passed a method reference parameter.

```
method "test"(&$inReferenceParam)
  $local := 13
  writebr(variable name($local))
  write(variable name($inReferenceParam))
end method

$foo := 7
test($foo) // output is "$local<br />$foo"
```

If *inVariable* is not defined, an error is generated and execution is aborted.

Plugin Commands

In addition to the execute plugin commands used to invoke Active4D, Active4D provides a number of utility plugin commands:

A4D Abandon session

A4D Base64 decode

A4D Base64 encode

A4D Blowfish decrypt

A4D Blowfish encrypt

A4D FLUSH LIBRARY

A4D GET LICENSE INFO

A4D Get root

A4D GET SESSION DATA

A4D GET SESSION STATS

A4D Get time remaining

A4D Get version

A4D Import library

A4D MD5 sum

A4D Native to URL path

A4D RESTART SERVER

A4D SET ROOT

A4D STRIP 4D TAGS

A4D URL decode path

A4D URL decode query

A4D URL encode path

A4D URL encode query

A4D URL to native path

A4D Abandon session

v6.3

A4D Abandon session(inSessionID) → Number

Parameter	Type	Description
inSessionID	Number	→ Session internal ID

Discussion

This command is the equivalent of the Active4D **abandon session** command, but allows you to abandon any session given its internal id.

If the session is abandoned successfully, zero is returned. If there is an error, 1 is returned and an error message is logged to the Active4D log.

A4D Base64 decode

version 4.0

A4D Base64 decode(inData {; inCharset}) → Text

Parameter	Type	Description
inData	Text	→ Base64 encoded text
inCharset	Text	→ Charset to convert from
Result	Text	← Original data

Discussion

This command is the equivalent of the Active4D **base64 decode** command.

A4D Base64 encode

version 4.0

A4D Base64 encode(inData {; inCharset}) → Text

Parameter	Type	Description
inData	Text	→ Original data
inCharset	Text	→ Charset to convert to
Result	Text	← Encoded text

Discussion

This command is the equivalent of the Active4D **base64 encode** command.

A4D Blowfish decrypt

version 4.0

A4D Blowfish decrypt(inBlobData; inPassphrase {; inIV {; inCharset}}) → Text

Parameter	Type	Description
inBlobData	BLOB	→ Encrypted data
inPassphrase	Text	→ Key used to encrypt data
inIV	Text	→ Seed text used to encrypt data
inCharset	Text	→ Charset to convert from
Result	Text	← Decrypted text

Discussion

This command is the equivalent of the Active4D command **blowfish decrypt**.

A4D Blowfish encrypt

**version 4.0
modified v5**

A4D Blowfish encrypt(inText; inPassphrase {; inIV {; inCharset}}) → BLOB

Parameter	Type	Description
inText	Text	→ Text to encrypt
inPassphrase	Text	→ Key used to encrypt text
inIV	Text	→ Seed text used for encryption
inCharset	Text	→ Character set to convert to
Result	BLOB	← Encrypted data

Discussion

This command is the equivalent of the Active4D command **blowfish encrypt**.

A4D FLUSH LIBRARY

**version 1
modified v5**

A4D FLUSH LIBRARY(inLibName)

Parameter	Type	Description
inLibName	Text	→ The name of the library to flush

Discussion

This command purges the specified library from memory. The next import of the specified library will reload the library.

If *inLibName* is "*", all libraries are flushed from memory. If *inLibName* "@", all libraries except the Active4D library are flushed from memory.

When you call **A4D FLUSH LIBRARY**, the library is not flushed immediately, because libraries cannot be flushed until all Active4D interpreters stop executing. Rather, a request to flush the given library is passed to the housekeeper process. The next time the housekeeper runs, it will wait for all interpreters to stop executing, then it will flush the library (or libraries).

Because of this behavior, if **A4D FLUSH LIBRARY** is called again before the previous flush request is fulfilled, the second flush request will replace the first one. Ordinarily this should not be a problem, since you should not need to call **A4D FLUSH LIBRARY** at all.

If *inLibName* is "*" or "Active4D", the Active4D library will be flushed and immediately reloaded, but the **On Application Start** event handler will not be run again. You have to execute **A4D RESTART SERVER** to accomplish that.

In general, you should not flush the Active4D library. If you are developing the event handlers in the Active4D library, you should create an auxiliary library (e.g. _active4D.a4l) and pass the Active4D event handler calls to that library. When you modify the auxiliary library, it will be reloaded and the Active4D library will remain unaffected.

Ordinarily you would have no need to use this command. If the "auto refresh libs" option is on in Active4D.ini, libraries are flushed and reloaded automatically when they are modified. If the "auto refresh libs" option is off, you must use this command to flush the library in order to reload it.

A4D Get IP address

v5

A4D Get IP address → Text

Parameter	Type	Description
Result	Text	← IP address of host

Discussion

This command returns the first IP address on the first ethernet interface on the host machine.

A4D Get MAC address

v5

A4D Get MAC address → Text

Parameter	Type	Description
Result	Text	← MAC address of host

Discussion

This command returns the MAC (Media Access Control) address of the first network interface on the host machine.

A4D Get root

version 1

A4D Get root → Text

Parameter	Type	Description
Result	Text	← Full path to the web root folder

Discussion

This command is the equivalent of the Active4D **get root** command.

A4D GET SESSION DATA

version 2

A4D GET SESSION DATA(inSessionID; outKeys; outTypes; outValues)

Parameter	Type	Description
inSessionID	Longint	→ Internal session ID to query
outKeys	String/Text Array	← Receives the item keys
outTypes	String/Text Array	← Receives the item types
outValues	Text Array	← Receives the item values

Discussion

This command retrieves information about all of the items in the session with the internal session ID *inSessionID* (as returned by the command **session internal id**). This command is used by the Active4D Session Monitor dialog.

If the arrays are not of the correct type, *OK* is set to zero and the arrays are left untouched.

If there is no current session or the session ID is invalid, the arrays are emptied and *OK* is set to zero.

The value of an item is represented in textual form. *Picture* and *BLOB* items are returned as the size of the item in bytes. Arrays are returned as a CR-delimited list of items, as would be returned by the command **join array**(\$array; "\r"; 0; true; true).

A4D GET SESSION STATS

version 2

A4D GET SESSION STATS(outIDs; outTimeouts; outRemainTimes; outSizes)

Parameter	Type	Description
outIDs	Longint Array	← Receives internal session IDs
outTimeouts	Real Array	← Receives session timeouts in minutes
outRemainTimes	Longint Array	← Receives seconds remaining till sessions timeout
outSizes	Longint Array	← Receives the memory used by the sessions in bytes

Discussion

This command retrieves information about all current sessions. This command is used by the Active4D Session Monitor dialog.

If any of the arrays are not of the type specified, they are all sized to zero and *OK* is set to zero. If the command succeeds, *OK* is set to 1.

Sessions which have expired but have not yet been purged (*zombies*) are returned with a remaining time of zero.

A4D GET LICENSE INFO

version 2

A4D GET LICENSE INFO(outUserName; outCompany; outLicenseType; outLicenseVersion; outServerIP; outExpirationDate; outPlatform {; outKeyFilePath})

Parameter	Type	Description
outUserName	Text	← The licensed user
outCompany	Text	← The licensed company
outLicenseType	Longint	← The type of license
outLicenseVersion	Text	← Active4D version licensed for
outServerIP	Text	← IP address for a regular deployment license, empty otherwise
outExpirationDate	Date	← Date a deployment license expires
outPlatform	Longint	← Always 3 (Mac and Windows)
outKeyFilePath	Text	← Full path to key file

Discussion

This command is the equivalent of the Active4D command **get license info**.

A4D Get MAC address

version 4.5

A4D Get MAC address → Text

Parameter	Type	Description
Result	Text	← MAC address

Discussion

This command returns the MAC address (ethernet ID) of the first network interface on the host machine.

A4D Get time remaining

version 2

A4D Get time remaining → Longint

Parameter	Type	Description
Result	Real	← Seconds till license timeout

Discussion

This command is the equivalent of the Active4D command **get time remaining**.

A4D Get version

version 1

A4D Get version → Text

Parameter	Type	Description
Result	Text	← Current version of Active4D

Discussion

This command returns a string describing the current version of Active4D.

A4D Import library

version 2

A4D Import library(inLibName; outErrorMessage) → Longint

Parameter	Type	Description
inLibName	Text	→ The name of the library to import
outErrorMessage	Text	← Contains any parsing errors that may occur
Result	Longint	← HTTP status code

Discussion

This command is the equivalent of the Active4D **import** command.

If a parsing error occurs during the loading of the library, a status of *500 (Internal server error)* will be returned, and *outErrorMessage* will contain information on the error.

If the library loads successfully, a status of *200 (OK)* will be returned and *outErrorMessage* will be empty.

A4D LOG MESSAGE

version 2

A4D LOG MESSAGE(inMessage {; inIsError})

Parameter	Type	Description
inMessage	Text	→ Message to append to the log file
inIsError	Boolean	→ True to mark as an error message

Discussion

This command is the equivalent of the Active4D **log message** command.

A4D MD5

version 2

A4D MD5(inData {; inCharset}) → Text

Parameter	Type	Description
inData	Text	→ The data to checksum
inCharset	Text	→ Charset to convert to
Result	Text	← 32-character hex checksum

Discussion

This command is the equivalent of the Active4D **md5 sum** command.

A4D Native to URL path

version 2

A4D Native to URL path(inPath) → Text

Parameter	Type	Description
inPath	Text	→ Path in native format
Result	Text	← Same path in URL format

Discussion

This command is the equivalent of the Active4D command **native to url path**.

A4D RESTART SERVER

version 2

A4D RESTART SERVER

Discussion

This command restarts the Active4D HTTP server, flushing all libraries, abandoning all sessions, and reloading all configuration files.

A4D Set HTTP body callback

v5

A4D Set HTTP body callback(inMethod) → Number

Parameter	Type	Description
inMethod	Text	→ Name of callback method
Result	Number	← 1 if method is found

Discussion

When called via On Web Connection, if Active4D receives a POST or PUT request, it calls a callback method to retrieve the body of the request. This command sets the method that Active4D calls.

Note: The callback method must be in the host database, not in a component.

If the callback method is found, 1 is returned. Otherwise zero is returned.

The callback method should have the following code:

```
C_BLOB($0)
GET HTTP BODY($0)
```

A4D SET ROOT

version 2

A4D SET ROOT(inPath)

Parameter	Type	Description
inPath	Text	→ Path to web root

Discussion

This command sets the web root directory. *inPath* may be an absolute or relative path in either native or URL (Unix) format. If it is a relative path, it is relative to the default 4D directory. For information on the default directory, see “The Default Directory” on page 42.

The path may also use any of the directory tokens available to paths in Active4D.ini. For more information on the directory tokens, see “Path Format” on page 43.

If *inPath* is a valid path, *OK* is set to 1. If it is not valid, *OK* is set to zero.

A4D STRIP 4D TAGS

version 3.0
deprecated v5

A4D STRIP 4D TAGS(ioBLOB)

Parameter	Type	Description
ioBLOB	BLOB	↔ BLOB to strip

Discussion

This command is deprecated and does nothing. It was only necessary with 4D 2003.

A4D URL decode path

version 2

A4D URL decode path(inText) → Text

Parameter	Type	Description
inText	Text	→ Text to decode
Result	Text	← Decoded text

Discussion

This command is the equivalent of the Active4D command **url decode path**.

A4D URL decode query

version 2

A4D URL decode query(inText) → Text

Parameter	Type	Description
inText	Text	→ Text to decode
Result	Text	← Decoded text

Discussion

This command is the equivalent of the Active4D command **url decode query**.

A4D URL encode path

version 2

A4D URL encode path(inText) → Text

Parameter	Type	Description
inText	Text	→ Text to encode
Result	Text	← Encoded text

Discussion

This command is the equivalent of the Active4D command **url encode path**, but it does not use the inUseAmpEntity parameter.

A4D URL encode query

version 2

A4D URL encode query(inText) → Text

Parameter	Type	Description
inText	Text	→ Text to encode
Result	Text	← Encoded text

Discussion

This command is the equivalent of the Active4D command **url encode query**.

A4D URL to native path

version 2

A4D URL to native path(inPath) → Text

Parameter	Type	Description
inPath	Text	→ Path in URL format
Result	Text	← Same path in native format

Discussion

This command is the equivalent of the Active4D command **url to native path**.

CHAPTER 12

Standard Libraries

Active4D extends its native command set by including a number of standard libraries that provide you with additional power and flexibility when creating your websites.

The standard libraries may be placed in any Active4D folder in the library search path. Currently, the standard libraries are:

- a4d.console
- a4d.debug
- a4d.json
- a4d.lists
- a4d.utils
- a4d.web
- Batch
- Breadcrumbs
- fusebox
- fusebox.conf
- fusebox.head
- RowSet
- SessionHandler

Using the Standard Libraries

To use a library, you can either **import** it and then call its methods or reference its constants, or simply use the full *library.method* syntax, in which case the library is implicitly imported if necessary.

The methods in these libraries are designed to make common programming tasks *much* easier. It will pay many times over for you to learn and use them.

Note: The SessionHandler library is imported automatically if you configure it as the “session handler” option in Active4D.ini.

a4d.console

This library is composed of methods that dump various data structures and internal information to the Active4D debug console. For more information on the debug console, see “The Active4D Debugging Console” on page 606.

Ordinarily you will want to use the methods in *a4d.debug* to dump debugging information, as the formatting is considerably easier to read than what is output by the methods in *a4d.console*.

However, there are times when you cannot dump information to a web page via the methods in *a4d.debug*. For example, there may be an execution error that is preventing the page from displaying. In such cases you can replace the method you would normally use in *a4d.debug*, such as *dump collection*, with the same method in *a4d.console*.

clear**v6.0**

clear

Discussion

This method clears the debug console.

dump array**version 4.0
modified v6.1r6**

```
dump array(inArray {}; inDisplayInline {}; inName {}; inDisplayCollections {}; inShowType
           {}; inShowZeroElement {}; inBlobTextFormat{}))
```

Parameter	Type	Description
inArray	Array	→ Array to dump
inDisplayInline	Boolean	→ If True, array is displayed on one line
inName	Text	→ Name shown in header of dump
inDisplayCollections	Boolean	→ If True collection handles may be displayed as collections
inShowType	Boolean	→ True to show type of items in collections
inShowZeroElement	Boolean	→ If True, show the zero element
inBlobTextFormat	Number	→ Format of text in BLOBs

Discussion

This method writes the contents of the given array to the debug console. The name of the array referred to by *inArray* and its type is displayed in the header. If *inName* is passed in, that is used instead of the source array name.

If *inDisplayInline* is passed and is *True*, the current element of the array is displayed, followed by the elements of the array on one line, separated by commas.

Otherwise the element number and element value are displayed in two columns. If the array is a String or Text array, the element values are surrounded by double quotes. If the array is a Longint or Real array and *inDisplayCollections* is passed and is *True*, array elements that are valid collection handles will be displayed as collections by calling **dump collection**. If *inShowType* is *True*, the collection items will be displayed with their type.

If *inShowZeroElement* is *False* (the default), the zero element of the array is not displayed.

If the array is a BLOB array and contains text, you may pass the format of the text within the BLOB (e.g. *UTF8 text without length*) in *inBlobTextFormat*. If *inBlobTextFormat* is passed and is ≥ 0 , the content of each element is displayed as text. Otherwise the number of bytes in each element of the array is displayed.

dump collection

version 4.0
modified v6.1r6

```
dump collection(inCollection; { inName {; inShowType {; inFilter {; inBlobTextFormat}}})
```

Parameter	Type	Description
inCollection	Collection Iterator	→ Iterator or handle of collection to dump
inName	Text	→ Display name of collection
inShowType	Boolean	→ True to display types of values
inFilter	Text	→ Regexp to filter the results
inBlobTextFormat	Number	→ Format of text in BLOBs

Discussion

This method dumps the contents of the given collection to the debug console. The key and value of every item in the collection are displayed, and if *inShowType* is provided and is *True*, the value type is displayed after the key.

If *inName* is passed, it is displayed in the header. If it is not passed, the name of the variable passed as *inCollection* is displayed in the header.

If an item value is an array, the array is displayed as if **dump array**(\$array; False; ""; True) is called.

If *inFilter* is passed and the first character is "-", the rest of *inFilter* is used as a regular expression and matching items are hidden. Otherwise only items matching the regular expression in *inFilter* are displayed.

If an item of the collection is a BLOB or BLOB array and contains text, you may pass the format of the text within the BLOB (e.g. *UTF8 text without length*) in *inBlobTextFormat*. If *inBlobTextFormat* is passed and is ≥ 0 , the content of each BLOB is displayed as text. Otherwise the number of bytes in each BLOB is displayed.

Warning: If an item of a collection is itself a collection reference, this method is called recursively to display its contents. So if you have any circular collection references infinite recursion and a crash will result.

dump form variables

version 4.0
modified v6.0

dump form variables{{(inFilter)}}

Parameter	Type	Description
inFilter	Text	→ Regexp to filter the results

Discussion

This method calls the **dump collection** method, passing an iterator to the *form variables* collection.

dump license info

version 4.5

dump license info

Discussion

This method creates a formatted dump of the relevant license information contained in your current key file, as well as the full path to the key file.

dump query params

version 2
modified v6.0

dump query params{{(inFilter)}}

Parameter	Type	Description
inFilter	Text	→ Regexp to filter the results

Discussion

This method calls the **dump collection** method, passing an iterator to the *query params* collection.

dump request info

version 2
modified v6.0

dump request info{(inFilter)}

Parameter	Type	Description
inFilter	Text	→ Regexp to filter the results

Discussion

This method calls the **dump collection** method, passing an iterator to the *request info* collection.

dump session

version 2
modified v6.1r6

dump session{(inFilter {; inBlobTextFormat})}

Parameter	Type	Description
inFilter	Text	→ Regexp to filter the results
inBlobTextFormat	Number	→ Format of text in BLOB arrays

Discussion

This method calls the **dump collection** method, passing an iterator to the current session.

a4d.debug

This library is composed of methods that create a nicely formatted HTML dump of various data structures and internal information, suitable for viewing in a web page.

Ordinarily you will want to use the methods in *a4d.debug* to dump debugging information, as the formatting is considerably easier to read than what is output by the methods in *a4d.console*.

However, there are times when you cannot dump information to a web page via the methods in *a4d.debug*. For example, there may be an execution error that is preventing the page from displaying. In such cases you can replace the method you would normally use in *a4d.debug*, such as *dump collection*, with the same method in *a4d.console*.

dump array

version 4.0
modified v6.1r6

```
dump array(inArray {; inDisplayInline {; inName {; inDisplayCollections {; inShowType  
{; inShowZeroElement {; inBlobTextFormat}}}}})
```

Parameter	Type	Description
inArray	Array	→ Array to dump
inDisplayInline	Boolean	→ If True, array is displayed on one line
inName	Text	→ Name shown in header of dump
inDisplayCollections	Boolean	→ If True collection handles may be displayed as collections
inShowType	Boolean	→ True to show type of items in collections
inShowZeroElement	Boolean	→ If True, show the zero element
inBlobTextFormat	Number	→ Format of text in BLOBs

Discussion

This method writes the contents of the given array to the response buffer.

If *inName* is passed and is not "-", it is displayed in the header. If it is "-", the header is not displayed at all. If it is not passed, the name and type of the variable passed in *inArray* is displayed in the header.

If *inDisplayInline* is passed and is *True*, the current element of the array is displayed, followed by the elements of the array on one line, separated by commas.

Otherwise the element number and element value are displayed in two columns. If the array is a String or Text array, the element values are surrounded by double quotes. If the array is a Longint or Real array and *inDisplayCollections* is passed and is *True*, array elements that are valid collection handles will be displayed as collections by calling **dump collection**. If *inShowType* is *True*, the collection items will be displayed with their type.

If *inShowZeroElement* is *False* (the default), the zero element of the array is not displayed.

If the array is a BLOB array and contains text, you may pass the format of the text within the BLOB (e.g. *UTF8 text without length*) in *inBlobTextFormat*. If *inBlobTextFormat* is passed and is ≥ 0 , the content of each element is displayed as text. Otherwise the number of bytes in each element of the array is displayed.

dump collection

version 2
modified v6.1r6

```
dump collection(inCollection; { inName {; inShowType {; inInlineArrays {; inFilter {;
  inBlobTextFormat}}}}})
```

Parameter	Type	Description
inCollection	Collection Iterator	→ Iterator or handle of collection to dump
inName	Text	→ Display name of collection
inShowType	Boolean	→ True to display types of values
inInlineArrays	Boolean	→ True to display arrays inline
inFilter	Text	→ Regexp to filter the results
inBlobTextFormat	Number	→ Format of text in BLOBs

Discussion

This method writes a formatted dump of the given collection to the response buffer. The key and value of every item in the collection are displayed in two columns, and if *inShowType* is provided and is *True*, a third column displays the value type.

If *inName* is passed and is not "-", it is displayed in the header. If it is "-", the header is not displayed at all. If it is not passed, the name of the variable passed as *inCollection* is displayed in the header.

If an item value is an array, the display depends on the value of *inInlineArrays*. If *inInlineArrays* is not passed or is *True*, the current element of the array is displayed, followed by a comma-delimited list of the array elements, with text or string arrays having their elements surrounded by quotes. If *inInlineArrays* is *False*, arrays are displayed as if **dump array**(\$array; False; ""; True) is called.

If *inFilter* is passed and the first character is "-", the rest of *inFilter* is used as a regular expression and matching items are hidden. Otherwise only items matching the regular expression in *inFilter* are displayed.

If an item of the collection is a BLOB or BLOB array and contains text, you may pass the format of the text within the BLOB (e.g. *UTF8 text without length*) in *inBlobTextFormat*. If *inBlobTextFormat* is passed and is ≥ 0 , the content of each BLOB is displayed as text. Otherwise the number of bytes in each BLOB is displayed.

Warning: If an item of a collection is itself a collection reference, this method is called recursively to display its contents. So if you have any circular collection references infinite recursion and a crash will result.

dump configuration

version 2

```
dump configuration{(inWhich)}
```

Parameter	Type	Description
inWhich	Text	→ Type of configuration to dump

Discussion

This method dumps the contents of the entire built in **configuration** collection, or if *inWhich* is passed, only that particular configuration item.

For a list of configuration items, see “configuration” on page 333.

dump form variables

version 2

```
dump form variables{(inFilter)}
```

Parameter	Type	Description
inFilter	Text	→ Regexp to filter the results

Discussion

This method calls the **dump collection** method, passing an iterator to the *form variables* collection.

This is the preferred method of inspecting the form variables posted to a page.

dump license info

**version 3
modified version 4.0**

```
dump license info
```

Discussion

This method creates a formatted dump of the relevant license information contained in your current key file, as well as the full path to the key file.

dump locals

version 3
modified v6.1r6

```
dump locals{(inDumpChain {; inFilter {; inBlobTextFormat{}}}
```

Parameter	Type	Description
inDumpChain	Boolean/Text	→ Determines which scopes should be dumped
inFilter	Boolean/Text	→ Filters the output
inBlobTextFormat	Number	→ Format of text in BLOBs

Discussion

This command creates a formatted dump of the current local variables.

If *inDumpChain* is not passed or is *False* (the default), **dump locals** only displays the local variables for the current scope. If *inDumpChain* is passed and is *True*, all scopes in the call chain are dumped in succession, starting with the innermost scope and proceeding towards the top level scope. If *inDumpChain* is passed and is text, only the scope matching that name is dumped. If no matching scopes are found, all scopes are dumped.

If *inFilter* is not passed or is *False* (the default), **dump locals** does not display the fusebox core and its variables. If *inFilter* is passed and is *True*, all of the fusebox internals will be displayed. If *inFilter* is passed and is text, it is used as a regular expression to filter which locals are displayed. If *inFilter* begins with "-", any locals whose name matches the regular expression following the "-" are excluded. If *inFilter* begins with any other character, only those locals whose name matches the regular expression are displayed.

If a local variable is a BLOB or BLOB array and contains text, you may pass the format of the text within the BLOB (e.g. *UTF8 text without length*) in *inBlobTextFormat*. If *inBlobTextFormat* is passed and is ≥ 0 , the contents of each BLOB is displayed as text. Otherwise the number of bytes in each BLOB is displayed.

This command is especially useful for getting a quick snapshot of the local variable state at a given point in the script execution.

dump query params

version 2
modified v6.0

```
dump query params{(inFilter)}
```

Parameter	Type	Description
inFilter	Text	→ Regexp to filter the results

Discussion

This method calls the **dump collection** method, passing an iterator to the *query params* collection.

This is the preferred method of inspecting the query string in the request URL.

dump request version 2 modified v6.0

dump request{(inFilter)}

Parameter	Type	Description
inFilter	Text	→ Regexp to filter the results

Discussion

This method calls **dump query params**, **dump form variables**, **dump request info**, and **dump session**.

dump request info version 2 modified v6.0

dump request info{(inFilter)}

Parameter	Type	Description
inFilter	Text	→ Regexp to filter the results

Discussion

This method calls the **dump collection** method, passing an iterator to the *request info* collection.

This is the preferred method of inspecting the HTTP headers sent to a page.

dump selection version 4.0

dump selection(inTable {; inFields})

Parameter	Type	Description
inTable	Pointer	→ Pointer to table to dump
inFields	Text	→ Field list

Discussion

This method does a nicely formatted dump of the current selection of *inTable*. The current record within the selection is highlighted.

If *inFields* is not passed, all fields in the table are displayed.

If *inFields* is passed, it should be a semicolon delimited list of fields to display. Whitespace around the field name is ignored. If a field is in *inTable*, you do not need to specify the table name. You may also include fields from related one tables, in which case you must include the table name. If the field name is “#” (or “[table]#” for a related table), the current record number for the table will be displayed.

You may also specify a format for a field by putting a format expression after the field name, separated by a colon. For BLOB fields that contain text, the format expression should be one of the format constants (a number or a named constant) you would use with **BLOB to text**. For other fields, the format expression is what you would pass to the **String** command to format the field’s value. For numeric types, use a format string (without quotes). For date and time fields, you may use a number or a 4D named constant such as “HH MM SS”.

Note: Because semicolon is used as the field delimiter, you may not use semicolons in the field format.

Examples

The following code will display all fields for the [ingredients] table:

```
a4d.debug.dump selection(->[ingredients])
```

The following code will display the id, name, and price fields from [ingredients], as well as the related one [vendors]name field. In addition, the price field will be formatted with the format string “\$###.00”:

```
$fields := "id;name;price:$###.00;[vendors]name"
a4d.debug.dump selection(->[ingredients]; $fields)
```

dump session

version 2
modified v6.0

```
dump session{{(inInlineArrays {}; inFilter {}; inBlobTextFormat)}}}
```

Parameter	Type	Description
inInlineArrays	Boolean	→ True to display arrays inline
inFilter	Text	→ Regexp to filter the results
inBlobTextFormat	Number	→ Format of text in BLOBs

Discussion

This method calls the **dump collection** method, passing an iterator to the current session.

This is the preferred method of inspecting session values.

dump session stats**version 3**

dump session stats {(inSortBy {; inDir}}}

Parameter	Type	Description
inSortBy	Text	→ Column to sort by
inDir	Text	→ '>' to sort ascending, '<' descending

Discussion

This method creates a formatted dump of all current sessions, displaying the following information, one row for each session:

Column	Description
ID	Session internal id
Timeout	Session timeout
Time Remaining	Time before session expires, zero means expired
Size	Total bytes of memory used by session

The rows may be sorted by passing in one of the column names in *inSortBy*. If no name or an invalid name is passed in, the rows will be sorted by the time remaining. The sort direction is determined by *inDir*. If *inDir* is not passed or is not valid, the rows will be sorted in ascending order.

Sessions which have expired but have not yet been purged (*zombies*) will be displayed with a time remaining of "00:00:00".

a4d.json

If you are using an AJAX-based Javascript library in your web sites, at some point it is likely you will need to create JSON (see json.org) formatted data within Active4D.

Active4D provides a full set of commands for easily creating properly formatted JSON data from all of the data types Active4D supports, including collections. In addition, there are commands for quickly adding a selection of records or a RowSet to JSON data.

Note: The functionality of the a4d.json library has been replaced by the native JSON commands, which are *much* faster (up to 50x) and more powerful. For backwards compability, the a4d.json library has been retained, but is now just a thin wrapper around the built in JSON commands, resulting in much greater speed.

It is recommended that you convert any a4d.json-based code to use the built in commands, as this library will be deprecated in a future version.

Using the library

The *a4d.json* library has two primary interfaces: a high level interface and a low level interface. Most of the time you will use the high level interface to create, populate and output a JSON "object". Most of the high level methods names begin with "add". If you need more control over the format of the JSON data, you can use the low level methods, whose names begin with "encode".

UTF-8 and Ajax

When using Ajax requests, posted data is encoded as UTF-8 and UTF-8 encoded data is expected in return.

Active4D handles this transparently. When Active4D detects an Ajax request, it automatically decodes posted UTF-8 data into Unicode. If you use the *a4d.json* library to return data to an Ajax request, the data is automatically encoded as UTF-8.

new**version 4.5**

new{(inWrap)} → Object

Parameter	Type	Description
inWrap	Boolean	→ True to wrap the JSON data within an enclosing object
Result	JSON Object	← JSON object to be used with other methods in this library

Discussion

This method creates a new JSON “object” to which data can be added.

Note: This method has been deprecated in favor of the native **new json** command.

add**version 4.5**

add(self; inKey; inValue {; inFilter}) → Object

Parameter	Type	Description
self	JSON object	→ Object to add data to
inKey	Text	→ Key for data item
inValue	<any>	→ Data to add
inFilter	Text	→ Filter for collections
Result	JSON object	← self

Discussion

This method adds data to a JSON object.

Note: This method has been deprecated in favor of the native **add to json** command.

If *inKey* is not an empty string, a new keyed item is added to the object. If *inKey* is empty, data is added without a key. Ordinarily you would only add data without a key if you are dynamically building an array using the **startArray** and **endArray** methods.

The data in *inValue* is converted to JSON format according to its type:

Value type	Result
String/Text	Encoded string, double-quoted
Number	Number
Date	Date in IETF “Mon Day, Year” format

Value type	Result
Boolean	<i>true</i> or <i>false</i>
Collection	Encoded as a subobject
JSON object	Added as a subobject
Other	<i>null</i>

If *inFilter* is passed and *inValue* is a collection, it is used to filter the contents of *inValue*. For more information, see “[encodeCollection](#)” on page 464.

Like all of the *a4d.json* methods, this method returns *self*, which allows you to chain several calls together on one line.

Examples

To understand how this method works, let’s look at some examples to see the resulting output. First let’s look at the output from the basic data types:

```
$json := a4d.json.new
$json->add("name"; "Sri Chinmoy")->add("age"; 76)
$json->add("birthdate"; !08/27/1931!)->add("Bengali"; true)
$json->add("time"; ?07:13:27?)
$json->write

// The output is:
{"name":"Sri Chinmoy","age":76,"birthdate":"Aug 27, 1931",
 "Bengali":true,"time":null}
```

Notice that the “time” item returned a value of *null*, because time is not a supported type in JSON. Also notice how we could chain multiple calls to **add** together.

Note: If you are confused by the `->` notation in the above example, see “[Creating a Poor Man’s Class](#)” on page 124.

Now let’s look at what happens if we add a collection to a JSON object.

```
$json := a4d.json.new
$json->add("name"; "Sri Chinmoy")
$info := new collection("age"; 76; "birthdate"; !08/27/1931!; \
    "Bengali"; true; "time"; ?07:13:27?)
array text($info{"siblings"}; *; "Chitta"; "Mantu"; \
    "Hriday"; "Lily"; "Arpita"; "Ahana")
$json->add("info"; $info)
$json->write

// The output is:
{"name":"Sri Chinmoy","info":{"age":76,"birthdate":"Aug 27,
1931","Bengali":true,"time":null,"siblings":["Chitta","Mantu",
"Hriday","Lily","Arpita","Ahana"]}}
```

As you can see, the items in a collection are added as a subobject with the given key. The values in a collection are added with the **add** method, unless a value is an array, in which case it is added using the **addArray** method.

Next let's look at what happens if we add an existing JSON object to another JSON object.

```
$json := a4d.json.new
$json->add("name"; "Sri Chinmoy")
$info := a4d.json.new
$info->add("age"; 76)
$info->add("birthdate"; !08/27/1931!)
$info->add("Bengali"; true)
$info->add("time"; ?07:13:27?)
$json->add("info"; $info)
$json->write

// The output is:
{"name":"Sri Chinmoy","info":{"age":76,"birthdate":"Aug 27,
1931","Bengali":true,"time":null}}
```

Finally let's look at an example of adding a value with no key.

```
array text($siblings; *; "Chitta"; "Mantu"; \\  
          "Hriday"; "Lily"; "Arpita"; "Ahana")

$json := a4d.json.new
$json->add("name"; "Sri Chinmoy")
$json->startArray("siblings")

for each($siblings; $name)
  $json->add(""; $name)
end for each

$json->endArray
$json->write

// The output is:
{"name":"Sri Chinmoy","siblings":["Chitta","Mantu","Hriday",  
"Lily","Arpita","Ahana"]}
```

addArray

version 4.5

addArray(self; inKey; inArray) → Object

Parameter	Type	Description
self	JSON object	→ Object to add data to
inKey	Text	→ Key for data item
inArray	Array	→ Array to add
Result	JSON Object	← self

Discussion

This method adds array data to a JSON object.

Note: This method has been deprecated in favor of the native **add to json** command.

If *inKey* is not an empty string, a new keyed item is added to the object. If *inKey* is empty, data is added without a key. Ordinarily you would only add data without a key if you are dynamically building an array using the **startArray** and **endArray** methods.

The elements of *inArray* are converted to JSON format according to the array type:

Value type	Result
ARRAY STRING/TEXT	Encoded string, double-quoted
ARRAY INTEGER/LONGINT/REAL	Number
ARRAY DATE	Date in IETF “Mon Day, Year” format
ARRAY BOOLEAN	<i>true</i> or <i>false</i>

Like all of the *a4d.json* methods, this method returns *self*, which allows you to chain several calls together on one line.

Note: For array types other than those listed above, this method will return an empty JSON array.

Examples

```
array string(27; $names; *; "Tom"; "Dick"; "Harry")
array longint($ages; *; 13; 27; 31)
array picture($picts; 3)
$json := a4d.json.new
$json->addArray("names"; $names)->addArray("ages"; $ages)
$json->addArray("oops"; $picts)
$json->write

// Output
{"names":["Tom", "Dick", "Harry"], "ages":[13, 27, 31], "oops":[]}
```

Note that the picture array was encoded as an empty JSON array since pictures are not a supported data type.

addDateTime

version 4.5

addDateTime(self; inKey; inDate; inTime; inTimezone) → Object

Parameter	Type	Description
self	JSON object	→ Object to add data to
inKey	Text	→ Key for data item
inDate	Date	→ Date portion of datetime
inTime	Time	→ Time portion of datetime
inTimezone	Number	→ Timezone portion of datetime
Result	JSON Object	← self

Discussion

This method adds an IETF format datetime item to a JSON object. Such an item can be turned into a Javascript Date object by passing the item string to the Date constructor.

Note: This method has been deprecated in favor of the native **add datetime to json** command.

If *inKey* is not an empty string, a new keyed item is added to the object. If *inKey* is empty, a datetime item is added without a key. Ordinarily you would only add a datetime without a key if you are dynamically building an array using the **startArray** and **endArray** methods.

inTimezone should be in the timezone indicated in the *inTimezone* parameter. *inTimezone* should be minute offset from GMT.

Like all of the *a4d.json* methods, this method returns *self*, which allows you to chain several calls together on one line.

Example

```
$json := a4d.json.new
// add a datetime in EST (GMT-5)
$json->addDateTime("DA"; !02/17/1980!; ?20:31:07?; -5 * 60)
$json->write

// The output is:
{"DA":"Feb 17, 1980 20:31:07 GMT-0500"}

// Javascript on the client, JSON is in a variable called json
var da = new Date(json.DA)
```

addFunction

version 4.5

addFunction(self; inName; inBody) → Object

Parameter	Type	Description
self	JSON object	→ Object to add data to
inName	Text	→ Function name
inBody	Text	→ Function body
Result	JSON Object	← self

Discussion

This method adds an item to *self* with the key *inName* and a value of *inBody* as is, unquoted and unencoded. This is useful when you need to add a function (such as a handler or callback) to your JSON data.

Note: This method has been deprecated in favor of the native **add function to json** command.

Like all of the *a4d.json* methods, this method returns *self*, which allows you to chain several calls together on one line.

Example

```
$json := a4d.json.new
$json->addFunction("renderer"; \\  
"function(value, metadata, record) {  
    return record.data.title + " " + value;  
}"")
```

addRowSet**version 4.5**

`addRowSet(self; inRowSet {}; inCountKey {}; inDataKey {}; inMap
{; inFirst {}; inLimit}})) → Object`

Parameter	Type	Description
<code>self</code>	JSON object	→ Object to add data to
<code>inRowSet</code>	RowSet	→ RowSet from which to get data
<code>inCountKey</code>	Text	→ Key for row count item
<code>inDataKey</code>	Text	→ Key for row data item
<code>inMap</code>	Text	→ JSON name to RowSet column map
<code>inFirst</code>	Number	→ Index of first row to add
<code>inLimit</code>	Number	→ Maximum number of rows to add
<code>Result</code>	JSON Object	← self

Discussion

This method adds rows from *inRowSet* to the JSON object *self*. If you have a RowSet and you want to use it for generating JSON data, this method is the fastest and easiest way to do so. If the RowSet is selection-based and is not being used for other purposes, in most cases you will want to use **addSelection** instead of this method.

Note: This method has been deprecated in favor of the native **add rowset to json** command.

The RowSet's rows are added as an array of objects, with each object containing one item for each column of data. The column data is converted as if were passed to the **add** method. If *inDataKey* is passed and is non-empty, the array will have the key *inDataKey*.

If *inCountKey* is passed and is non-empty, an item will be added to *self* whose key is *inCountKey* and whose value is the number of rows in *inRowSet*.

Note: The number of rows returned with *inCountKey* is the result of calling *\$inRowSet->rowCount*, not *\$inRowSet->sourceRowCount*.

If *inMap* is not passed or is empty, all of the columns in *inRowSet* will be added to the row array. If *inMap* is passed and is non-empty, it must be a semicolon-delimited list of mappings from RowSet column names to JSON key names. If the RowSet column name will be used as is, it is sufficient to use just the column name. If you want to rename a RowSet column, then the mapping should be a `<JSON key>:<RowSet column>` pair. This allows you to specify a subset of the RowSet columns for inclusion in the JSON data, and/or to rename the RowSet columns.

If *inFirst* is passed and is ≥ 1 , it specifies the one-based index of the first row from *inRowSet* that will be added to *self*. If *inLimit* is passed and is ≥ 0 , it specifies the maximum number of rows from *inRowSet* that will be added to *self*. Together, *inFirst* and *inLimit* make it easy to specify a subset of rows, which is typically the case when paging through a large RowSet.

Note: Depending on the Ajax toolkit you are using, it is likely that the start index for a page of data will be zero-based. It is up to you to add 1 to make it one-based before using the value for *inFirst*.

Like all of the *a4d.json* methods, this method returns *self*, which allows you to chain several calls together on one line.

Examples

Let's look at a few examples to illustrate the typical use of the various options. First, we'll create a RowSet and then add it a JSON object.

```
// Assume we have a selection of 3 [employee] records
$map := ""
name: `concat(" "; [employee]first; [employee]last)`;
id: [employee]id;
birthdate: [employee]birthdate""
$rs := RowSet.newFromSelection(->[employee]; $map)
$json := a4d.json.new
$json->addRowSet($rs; "count"; "rows")
$json->write

// The output is:
{"count":3,"rows":[{"name":"Tiny Tim","id":31,"birthdate":"Apr
12, 1932"}, {"name":"James Taylor","id":27,"birthdate":"Mar 12,
1948"}, {"name":"Pat Metheny","id":13,"birthdate":"Aug 12,
1954"}]}
```

Now let's add the rest of the parameters to see the effect. This time we want to eliminate the *id* column from the JSON data, and we want to rename the RowSet *birthdate* column to the JSON key *dob*. In addition, we are receiving the starting index in a query parameter

called "start" whose value is "2", and the number of rows to return in a query parameter called "size" whose value is "1".

```
// RowSet setup is the same as the example above

// Use "name" column as is, rename "birthdate" to "dob"
$jmap := "name;dob:birthdate"

// If "start" query param is not passed, default to "0",
// convert it to one-based number
$first := num($attributes{"start"} | "0") + 1

// If "size" query param is not passed, default to "20"
$limit := num($attributes{"size"} | "20")
$json := a4d.json.new
$json->addRowSet($rs; "count"; "rows"; $jmap; $first; $limit)
$json->write

// The output is:
{"count":3,"rows":[{"name":"Pat Metheny","dob":"Aug 12, 1954"}]}
```

Note that the *count* item still returns 3, because the number of rows in the RowSet is still 3, even though there is only one row in the *rows* array. The reason for this is because when you are showing paging information for a RowSet, you usually want to display something like "Displaying records <start>-<end> of <total>", so you always need the total number of rows in the source dataset.

addSelection

version 4.5
modified v6.1

```
addSelection(self; inTable {; inCountKey {; inDataKey {; inMap
  {; inFirst {; inLimit}}}}) → Object
```

Parameter	Type	Description
self	JSON object	→ Object to add data to
inTable	Table pointer	→ Main table from which to get data
inCountKey	Text	→ Key for row count item
inDataKey	Text	→ Key for row data item
inMap	Collection	→ JSON name to value map
inFirst	Number	→ Index of first row to add
inLimit	Number	→ Maximum number of rows to add
Result	JSON Object	← self

Discussion

This method adds record data from the current selection of *inTable* to the JSON object *self*. If you have a selection of records and you want to use it for generating JSON data, this method is the fastest and easiest way to do so.

Note: This method has been deprecated in favor of the native **add selection to json** command.

This method operates in two modes:

- **selection mode:** If *inFirst* # -1, the selection's records are output as an array of objects, with each object containing one item for each field.

inFirst specifies the one-based index of the first record in the current selection of *inTable* that will be added to *self*. If *inLimit* is passed and is ≥ 0 , it specifies the maximum number of records from the current selection of *inTable* that will be added to *self*. Together, *inFirst* and *inLimit* make it easy to specify a subset of records, which is typically the case when paging through a large selection.

Note: Depending on the Ajax toolkit you are using, it is likely that the start index for a page of data will be zero-based. It is up to you to add 1 to make it one-based before using the value for *inFirst*.

- **current record mode:** If *inFirst* = -1, only the current record of the selection is output as a single object (not within an array) with one item for each field.

The field data is converted as if were passed to the **add** method. If *inDataKey* is passed and is non-empty, the record data will have the key *inDataKey*.

If *inCountKey* is passed and is non-empty, an item will be added to *self* whose key is *inCountKey* and whose value is the number of records in the current selection of *inTable*.

If *inMap* is not passed or is empty, all of the fields in *inTable* will be added to the record array, with the field name being the JSON column key. If *inMap* is passed and is non-zero, it must be a collection which maps JSON column keys to column values. The keys in *inMap* are used as the JSON column keys. The values may be one of three types:

- **Table pointer:** If the value is a table pointer, the current record number for *inTable* is output as a JSON number.
- **Field pointer:** If the value is a field pointer, the value of the field is output as if it were passed to the **add** method. Fields from tables other than *inTable* may be used if there is a many to one relation (it need not be automatic) between *inTable* and the foreign field's table. If any foreign fields are used in *inMap*, **RELATE ONE**(\$*inTable*->) is executed before each record is output to ensure related data is available.
- **Text:** If the value is text, it must be an expression than returns a value (although using **return** is not necessary). The resulting value is output as if it were passed to the **add** command. If you want the result to appended verbatim, without being JSON encoded, prefix the expression with "@". If you want to evaluate the expression in 4D instead of Active4D, prefix the expression (after "@") with "!".

If *inMap* contains an item whose key is "a4d.json.callback", then the value should be a text block of code to execute after each record of *inTable* is loaded. The code does not have to return a value. If you want the code to be executed in 4D instead of Active4D, prefix it with "!". Code executed in Active4D may consist of multiple statements separated by carriage returns. Code executed in 4D may only be one line.

Like all of the *a4d.json* methods, this method returns *self*, which allows you to chain several calls together on one line.

Examples

Let's look at a few examples to illustrate the typical use of the various options. First, we'll create a selection of records and then add it a JSON object.

```
// [employee] table
// id          Longint
// firstname   Alpha20
// lastname    Alpha20
// company_id  Longint, relate one with [company]id
// dob        Date
query([employee]; [employee]contact_id = $attributes{"id"})
$json := a4d.json.new
$json->addSelection(->[employee]; "count"; "rows")
$json->write

// The output is:
{"count":3,"rows":[{"id":31,"firstname":"Tiny","lastname":"Tim",
"company_id":101,"dob":"Apr 12, 1932"}, {"id":27,
"firstname":"James","lastname":"Taylor","company_id":107,
"dob":"Mar 12, 1948"}, {"id":13,"firstname":"Pat",
"lastname":"Metheny","company_id":107,"dob":"Aug 12, 1954"}]}
```

Notice how all of the fields were automatically included in the JSON output. Now let's add the rest of the parameters to see the effect.

To make the output more useful, we would like to do the following:

- Include the record number of the [employee] table
- Concatenate the first name and last name into a single name column
- Return the company name instead of its id
- Rename "dob" to "birthdate"

Note: For an example of how to use the *inFirst* and *inLimit* parameters, see the documentation for "addRowSet" on page 452.

We accomplish this by creating a map and passing it to **addSelection**:

```
// Selection setup is the same as the example above

$map := new collection

// output record number
$map{"recnum"} := ->[employee]

// output concatenation of firstname + " " + lastname
$map{"name"} := "concat(\" \"; [employee]firstname;
[employee]lastname)"

// output foreign related one field
$map{"company"} := ->[company]name

// rename a field
$map{"birthdate"} := ->[employee]dob

// use a callback
$map{"a4d.json.callback"} := \\
"query([family]; [family]employee_id = [employee]id)\r" + \\
"query([family]; & [family]type = 1)" // type 1 is child

$map{"num_children"} := "records in selection([family])"

$json := a4d.json.new
$json->addSelection(->[employee]; "count"; "rows"; $map)
$json->write

// The output is:
{"count":3,"rows":[{"recnum":207,"name":"Tiny Tim",
"company":"Tulips, Inc.,"birthdate":"Apr 12, 1932",
"num_children":0},{ "recnum":331,"name":"James Taylor",
"company":"Gorilla Corp.,"birthdate":"Mar 12, 1948",
"num_children":2}, {"recnum":713,"name":"Pat Metheny",
"company":"The Way Up","birthdate":"Aug 12, 1954",
"num_children":1}]}
```

startArray

version 4.5

`startArray(self; inKey) → Object`

Parameter	Type	Description
self	JSON object	→ Object to add data to
inKey	Text	→ Key for array item
Result	JSON object	← self

Discussion

This method adds an array start marker to a JSON object.

Note: This method has been deprecated in favor of the native **start json array** command.

Ordinarily you would only call this method if you are dynamically building an array using the **startArray** and **endArray** methods instead of using the **addArray** method.

Warning: To ensure valid JSON data, be sure to balance a call to this method with a call to **endArray**.

If *inKey* is not an empty string, the array start marker is preceded by an item key.

Like all of the *a4d.json* methods, this method returns *self*, which allows you to chain several calls together on one line.

Example

```
array text($siblings; *; "Chitta"; "Mantu"; \\  
          "Hriday"; "Lily"; "Arpita"; "Ahana")  
  
$json := a4d.json.new  
$json->startArray("siblings")  
  
for each($siblings; $name)  
  $json->add(""; $name)  
end for each  
  
$json->endArray  
$json->write  
  
// The output is:  
{ "siblings": ["Chitta", "Mantu", "Hriday", "Lily", "Arpita", "Ahana"] }
```

endArray

version 4.5

endArray(self) → Object

Parameter	Type	Description
self	JSON object	→ Object to add data to
Result	JSON object	← self

Discussion

This method adds an array end marker to a JSON object.

Note: This method has been deprecated in favor of the native **end json array** command.

Ordinarily you would only call this method if you are dynamically building an array using the **startArray** and **endArray** methods instead of using the **addArray** method.

Warning: To ensure valid JSON data, be sure to balance a call to this method with a previous call to **startArray**.

Like all of the *a4d.json* methods, this method returns *self*, which allows you to chain several calls together on one line.

Example

See “startArray” on page 458.

startObject

version 4.5

startObject(self; inKey) → Object

Parameter	Type	Description
self	JSON object	→ Object to add data to
inKey	Text	→ Key for subobject
Result	JSON object	← self

Discussion

This method adds an object start marker to a JSON object.

Note: This method has been deprecated in favor of the native **start json object** command.

Ordinarily you would only call this method if you are dynamically building an object using the **startObject** and **endObject** methods instead of using the **add** method.

Warning: To ensure valid JSON data, be sure to balance a call to this method with a call to **endObject**.

If *inKey* is not an empty string, the object start marker is preceded by an item key.

Like all of the *a4d.json* methods, this method returns *self*, which allows you to chain several calls together on one line.

Example

```
$json := a4d.json.new
$json->startArray("employees")

for each([employee])
  $json->startObject
  $json->add("name"; [employee]name)
  $json->add("age"; [employee]age)
  $json->endObject
end for each

$json->endArray
$json->write

// The output is:
{"employees": [{"name": "Tom", "age": 31}, {"name": "Dick", "age": 27}, {"name": "Harry", "age": 42}]}
```

endObject

version 4.5

endObject(self) → Object

Parameter	Type	Description
self	JSON object	→ Object to add data to
Result	JSON object	← self

Discussion

This method adds an object end marker to a JSON object.

Note: This method has been deprecated in favor of the native **end json object** command.

Ordinarily you would only call this method if you are dynamically building an object using the **startObject** and **endObject** methods instead of using the **add** method.

Warning: To ensure valid JSON data, be sure to balance a call to this method with a previous call to **startObject**.

Like all of the *a4d.json* methods, this method returns *self*, which allows you to chain several calls together on one line.

Example

See “startObject” on page 459.

toJSON

version 4.5

toJSON(self) → Text

Parameter	Type	Description
self	JSON object	→ Object from which to get JSON data
Result	Text	← JSON-formatted data

Discussion

This method returns the data that has been added to *self* as JSON-formatted text.

Note: This method has been deprecated in favor of the native **json to text** command.

Usually you will not need to call this method directly, since you will want to write the result of this method to the response buffer, and the **a4d.json.write** method does that for you.

Warning: If you do decide to write JSON data to the response buffer yourself (instead of using **a4d.json.write**), be sure to use the **write raw** command to prevent any HTML encoding.

write**version 4.5**

write(self; inSetContentType)

Parameter	Type	Description
self	JSON object	→ Object from which to write JSON data
inSetContentType	Boolean	→ True to set the content type

Discussion

This method writes the data that has been added to *self* as JSON-formatted text to the response buffer. This is the primary method you will use to return JSON data to a client.

Note: This method has been deprecated in favor of the native **write json** command.

If *inSetContentType* is not passed or is True, the content type of the response is set to "application/json".

writeln**version 4.5**

writeln(self)

Parameter	Type	Description
self	JSON object	→ Object from which to write JSON data
inCallback	String	→ Callback expected by JSONP request
inSetContentType	Boolean	→ True to set the content type

Discussion

This method writes the data that has been added to *self* as JSON-formatted text to the response buffer, wrapped in a call to the function *inCallback*. This is the primary method you will use to return JSON data to a client that is using the JSONP protocol.

Note: This method has been deprecated in favor of the native **write jsonp** command.

If *inSetContentType* is not passed or is True, the content type of the response is set to "application/json".

Example

Assume your front end makes a JSONP request with the following URL:

```
/enrollments/list?cb=CPJSONPConnectionCallbacks.callback32626
```

You build the JSON response and then return it like this:

```
$json := a4d.json.new
// add data to $json
$callback := $attributes{"cb"} // assuming fusebox
$json->writep($callback)
```

encode

version 4.5

encode(inValue {; inFilter}) → Text

Parameter	Type	Description
inValue	<any scalar value>	→ Value to encode
inFilter	Text	→ Filters collection items
Result	Text	← Encoded JSON data

Discussion

This method encodes a scalar (non-array) value for use in a JSON object. Ordinarily you would have no need to call this method directly, as it is used internally by the various **add** methods.

Note: This method has been deprecated in favor of the native **json encode** command.

Depending on the type of *inValue*, one of the more specific **encode<type>** methods will be called.

encodeArray

version 4.5

encodeArray(inArray) → Text

Parameter	Type	Description
inArray	Array	→ Array to encode
Result	Text	← Encoded JSON data

Discussion

This method encodes the items in *inArray* as a JSON array for use in a JSON object. Each item in the array is encoded by calling the appropriate encoder for the array type. If the

array type has no encoder, an empty array ("[]") is returned. For a list of the supported array types, see "addArray" on page 449.

Note: This method has been deprecated in favor of the native **json encode** command.

Ordinarily you would have no need to call this method directly, as it is used internally by the various **add** methods.

encodeBoolean

version 4.5

encodeBoolean(inBool) → Text

Parameter	Type	Description
inBool	Boolean	→ Value to encode
Result	Text	← Encoded JSON data

Discussion

This method encodes a boolean value for use in a JSON object. Ordinarily you would have no need to call this method directly, as it is used internally by the various **add** methods.

Note: This method has been deprecated in favor of the native **json encode** command.

The value returned will be either "true" or "false".

encodeCollection

version 4.5

encodeCollection(inCollection {; inFilter}) → Text

Parameter	Type	Description
inCollection	Collection	→ Collection to encode
inFilter	Text	→ Filters collection items
Result	Text	← Encoded JSON data

Discussion

This method encodes the data in *inCollection* as a subobject for use in a JSON object. Each key of the collection is encoded by calling **encodeString**, and each value is

encoded by calling the appropriate encoder for its type. If the type has no encoder, the value "null" is returned.

Note: This method has been deprecated in favor of the native **json encode** command.

Ordinarily you would have no need to call this method directly, as it is used internally by the various **add** methods.

If *inFilter* is passed and is non-empty, it is used as a matching expression to determine which items from *inCollection* are encoded. The rules for the filter expression are as follows:

- If the filter begins with "#", it performs an exclusion, i.e. all items that match the filter are excluded.
- Otherwise only items that match the filter are included.
- If the first character after the optional "#" is "/", it is considered a regular expression pattern and regex matching is done.
- Otherwise simple string comparison is performed.

Examples

Let's take a look at some simple filters to see how they affect the output of this method.

```
$c := new collection("foo"; 7; "bar"; !04/13/1964!; "baz"; false)

// simple string matching
$enc := a4d.json.encodeCollection($c; "foo")
// $enc = {"foo":7}

// simple string matching with wildcard
$enc := a4d.json.encodeCollection($c; "b@")
// $enc = {"bar":"Apr 13, 1964","baz":false}

// exclusion matching, simple string
$enc := a4d.json.encodeCollection($c; "#bar")
// $enc = {"foo":7,"baz":false}

// exclusion matching, simple string with wildcard
$enc := a4d.json.encodeCollection($c; "#b@")
// $enc = {"foo":7}

// regex matching
$enc := a4d.json.encodeCollection($c; "/foo|bar/")
// $enc = {"foo":7,"bar":"Apr 13, 1964"}

// exclusion matching, regex
$enc := a4d.json.encodeCollection($c; "#/foo|bar/")
// $enc = {"baz":false}
```

encodeDate

version 4.5

encodeDate(inDate) → Text

Parameter	Type	Description
inDate	Date	→ Value to encode
Result	Text	← Encoded JSON data

Discussion

This method encodes a date value for use in a JSON object. Ordinarily you would have no need to call this method directly, as it is used internally by the various **add** methods.

Note: This method has been deprecated in favor of the native **json encode** command.

The value returned will be in IETF Mon, Day Year format. For more information on date formatting, see “add” on page 446.

encodeString

version 4.5

encodeString(inString) → Text

Parameter	Type	Description
inString	Text	→ Value to encode
Result	Text	← Encoded JSON data

Discussion

This method encodes a string or text value for use in a JSON object. Ordinarily you would have no need to call this method directly, as it is used internally by the various **add** methods.

Note: This method has been deprecated in favor of the native **json encode** command.

The value returned is converted to UTF-8, surrounded with double quotes, and any non-printable characters are encoded according to JSON rules.

parse**v6.0r2
modified v6.1**

```
parse(inJSON {; inWantGlobalCollections {; inThrowOnError {; inDateKeys{}}) → <any>
```

Parameter	Type	Description
inJSON	Text	→ Text to parse
inWantGlobalCollections	Boolean	→ True to create global collections
inThrowOnError	Boolean	→ True to throw on malformed JSON
inDateKeys	Text	→ Regular expression to match keys for date conversion
Result	<any>	← Parsed JSON data

Discussion

This method parses JSON text into its corresponding Active4D value.

Note: This method has been deprecated in favor of the native **parse json** command.

All valid JSON types and syntaxes are supported, with the exception that array elements must all be of the same type. JSON types map to Active4D types as follows:

JSON type	Active4D type
object	collection
array	array
string	text
number	real
true/false	boolean
null	nil pointer

Note that JSON has no native representation for dates. To convert textual representations of dates into 4D dates, pass a delimited regular expression in *inDateKeys*. JSON strings or string arrays whose keys match the regular expression will be considered for conversion. Conversion occurs if the value matches one of the following patterns:

Pattern	Comments
Mmm d, yyyy	IETF date format. Mmm is a 3-letter English month abbreviation.
yyyy-mm-dd yyyy-mm-ddThh:mm:ss	ISO Date format

If *inWantGlobalCollections* is False (the default), all collections created from JSON objects will be local. If True, all collections will be global.

If *inThrowOnError* is True (the default), malformed JSON will throw an error.

Note: Because this method may return an array, you should always use the super assign operator (`::=`) to assign the result of this method.

convertJSONDates

v6.0r2

convertJSONDates(inObject {; inKeys}) → <any>

Parameter	Type	Description
inObject	<any>	→ Data to convert
inKeys	Text	→ Collection keys to consider for conversion
Result	<any>	← Converted data

Discussion

This method recursively traverses an object and converts text that looks like dates into 4D dates.

Note: This method has been deprecated in favor of the native **parse json** command, passing *inKeys* to the command.

Text will be converted if it matches one of the following patterns:

Pattern	Comments
Mmm d, yyyy	Format used by this library to convert 4D dates to JSON. Mmm is a 3-letter English month abbreviation.
yyyy-mm-dd	ISO Date format
yyyy-mm-ddThh:mm:ss	

Valid types for *inObject* are text, text array, longint array, and collection. Longint arrays are traversed, and if an element is a collection, this method will be called recursively with that collection.

For text arrays, if all of the elements of the array can be converted to dates, an array of the converted dates is returned. Otherwise the original text array is returned.

If *inKeys* is non-empty, it should be a delimited regular expression. When traversing collections, only collection keys which match will be considered for conversion. This allows for more efficient conversion with large collections, and also allows you to shield items from erroneous conversion.

The type of object returned depends on the type of *inObject*. Converted text arrays will be returned as date arrays, and converted text will be returned as a date. Otherwise the original object is returned.

Note: Because this method may return an array, you should always use the super assign operator (`::=`) to assign the result of this method.

a4d.lists

Lists are a way of putting multiple values into a single string, delimited by some character. This is very handy for some types of problems. For example, a URL can be treated as a list of path elements delimited by "/". Or the string "one,two,three" can be treated as a list of three elements. Or you can encode a list of values in a list and store it in a database text field.

Note the following attributes of lists:

- Empty elements are ignored, so "one,,three" would be a list of two elements, "one" and "three".
- Delimiters at the beginning and end of the list are ignored, so "/one/two/three/" would be a list of three elements.
- If the delimiter passed to a method contains more than one character, then any character in the delimiter can delimit the list elements, but only the first character in the delimiter will be used if the method adds an element to the list.
- List elements are numbered beginning at 1.
- The default delimiter for all list methods is ",".

append

version 3

```
append(inList; inValue {; inDelim}) → Text
```

Parameter	Type	Description
inList	Text	→ Delimited list
inValue	<any>	→ Value to append
inDelim	Text	→ List delimiter
Result	Text	← Modified list

Discussion

This method appends a value to a list. The value can be of any type that can be converted to text with the **String** command (which includes String and Text).

If *inDelim* is not passed, it defaults to "".

arrayToList

version 3

```
arrayToList(inArray {; inDelim}) → Text
```

Parameter	Type	Description
inArray	Array	→ Array to convert
inDelim	Text	→ List delimiter
Result	Text	← inArray as a delimited list

Discussion

This method converts *inArray* into a delimited list. If *inDelim* is not passed, it defaults to "".

Note: You can easily accomplish the same thing with the **join array** command.

changeDelims

version 3

changeDelims(inList; inNewDelim {; inDelim}) → Text

Parameter	Type	Description
inList	Text	→ Delimited list
inNewDelim	Text	→ New delimiter to use
inDelim	Text	→ List delimiter
Result	Text	← Modified list

Discussion

This method changes the delimiters in a list and returns a modified copy. If *inDelim* is not passed, it defaults to “,”.

Note: You can accomplish the same thing with **replace string**.

contains

version 3

contains(inList; inSubstr {; inDelim}) → Boolean

Parameter	Type	Description
inList	Text	→ Delimited list
inSubstr	Text	→ Text to match
inDelim	Text	→ List delimiter
Result	Boolean	← True if an element matches

Discussion

This method returns *True* if any element in the list contains a substring, doing a case- and diacritical-sensitive match.

If *inDelim* is not passed, it defaults to “,”.

containsNoCase

version 3

containsNoCase(inList; inSubstr {; inDelim}) → Boolean

Parameter	Type	Description
inList	Text	→ Delimited list
inSubstr	Text	→ Text to match
inDelim	Text	→ List delimiter
Result	Boolean	← True if an element matches

Discussion

This method returns *True* if any element in the list contains a substring, doing a case- and diacritical-insensitive match.

If *inDelim* is not passed, it defaults to *“,”*.

deleteAt

version 3

deleteAt(inList; inIndex {; inDelim}) → Text

Parameter	Type	Description
inList	Text	→ Delimited list
inIndex	Number	→ Element to delete
inDelim	Text	→ List delimiter
Result	Text	← Modified list

Discussion

This method deletes an element from a list. If *inDelim* is not passed, it defaults to *“,”*.

find

version 3

find(inList; inMatch {; inDelim}) → Number

Parameter	Type	Description
inList	Text	→ Delimited list
inMatch	Text	→ Text to match
inDelim	Text	→ List delimiter
Result	Number	← Index of first matching element

Discussion

This method returns the index of the first element of *inList* that exactly matches *inMatch*, doing a case- and diacritical-sensitive match. If no element matches, zero is returned.

If *inDelim* is not passed, it defaults to "".

findNoCase

version 3

findNoCase(inList; inMatch {; inDelim}) → Number

Parameter	Type		Description
inList	Text	→	Delimited list
inMatch	Text	→	Text to match
inDelim	Text	→	List delimiter
Result	Number	←	Index of first matching element

Discussion

This method returns the index of the first element of *inList* that matches *inMatch*, doing a case- and diacritical-insensitive match. If no element matches, zero is returned.

If *inDelim* is not passed, it defaults to "".

first

version 3

first(inList {; inDelim}) → Text

Parameter	Type		Description
inList	Text	→	Delimited list
inDelim	Text	→	List delimiter
Result	Text	←	First element of list

Discussion

This method returns the first element from a list, or an empty string if the list is empty. If *inDelim* is not passed, it defaults to "".

getAt

version 3

getAt(inList; inIndex {; inDelim}) → Text

Parameter	Type		Description
inList	Text	→	Delimited list
inIndex	Number	→	Element to get
inDelim	Text	→	List delimiter
Result	Text	←	Requested element of list

Discussion

This method returns an indexed element from a list, or an empty string if the list is empty or *inIndex* is out of range.

If *inDelim* is not passed, it defaults to `","`.

insertAt

version 3

insertAt(inList; inIndex; inValue {; inDelim}) → Text

Parameter	Type	Description
inList	Text	→ Delimited list
inIndex	Number	→ Where to insert
inValue	<any>	→ Value to insert
inDelim	Text	→ List delimiter
Result	Text	← Modified list

Discussion

This method inserts an element into a list and returns a modified copy. The value can be of any type that can be converted to text with the **String** command (which includes String and Text). If *inIndex* < 1, the element is prepended. If *inIndex* > **a4d.lists.len(inList)**, the element is appended.

If *inDelim* is not passed, it defaults to `","`.

last

version 3

last(inList {; inDelim}) → Text

Parameter	Type	Description
inList	Text	→ Delimited list
inDelim	Text	→ List delimiter
Result	Text	← Last element of list

Discussion

This method returns the last element from a list, or an empty string if the list is empty. If *inDelim* is not passed, it defaults to `","`.

len

version 3

```
len(inList {; inDelim}) → Number
```

Parameter	Type		Description
inList	Text	→	Delimited list
inDelim	Text	→	List delimiter
Result	Number	←	Number of elements in list

Discussion

This method returns the number of element in a list. If *inDelim* is not passed, it defaults to `','`.

listToArray

version 3

```
listToArray(inList; outArray {; inDelim})
```

Parameter	Type		Description
inList	Text	→	Delimited list
outArray	Array	←	Array to set
inDelim	Text	→	List delimiter

Discussion

This method converts a delimited list into an array. If *inDelim* is not passed, it defaults to `','`.

Note: You can accomplish the same thing with **split string**.

prepend

version 3

```
prepend(inList; inIndex; inValue {; inDelim}) → Text
```

Parameter	Type		Description
inList	Text	→	Delimited list
inValue	<any>	→	Value to insert
inDelim	Text	→	List delimiter
Result	Text	←	Modified list

Discussion

This method inserts an element at the beginning of a list and returns a modified copy. The value can be of any type that can be converted to text with the **String** command (which includes **String** and **Text**). If *inIndex* < 1, the element is prepended. If *inIndex* > **a4d.lists.len(inList)**, the element is appended.

If *inDelim* is not passed, it defaults to `","`.

qualify

version 3

`qualify(inList; inQualifier{; inDelim {; inCharAll{}}) → Text`

Parameter	Type	Description
<code>inList</code>	Text	→ Delimited list
<code>inQualifier</code>	Text	→ String to enclose with
<code>inDelim</code>	Text	→ List delimiter
<code>inCharAll</code>	Text	→ "char" or "all"
Result	Text	← New list

Discussion

This method encloses the elements of *inList* with *inQualifier* (e.g. double quotes).

If *inCharAll* is "char", the element will be skipped if it contains any digits. A modified copy of the list is returned.

If *inDelim* is not passed, it defaults to `","`.

rest

version 3

`rest(inList {; inDelim{}) → Text`

Parameter	Type	Description
<code>inList</code>	Text	→ Delimited list
<code>inDelim</code>	Text	→ List delimiter
Result	Text	← New list

Discussion

This method returns a copy of list starting from the second element. If *inList* has less than two elements, an empty string is returned.

If *inDelim* is not passed, it defaults to `","`.

setAt**version 3**

```
setAt(inList; inIndex; inValue {; inDelim}) → Text
```

Parameter	Type	Description
inList	Text	→ Delimited list
inIndex	Number	→ Element to set
inValue	<any>	→ Value to insert
inDelim	Text	→ List delimiter
Result	Text	← Modified list

Discussion

This method sets an element of a list and returns a modified copy. The value can be of any type that can be converted to text with the **String** command (which includes String and Text). If *inIndex* is out of range, nothing happens.

If *inDelim* is not passed, it defaults to "".

sort**version 3**

```
sort(inList {; inSortType {; inSortOrder {; inDelim}}}) → Text
```

Parameter	Type	Description
inList	Text	→ Delimited list
inSortType	Text	→ Not used, pass ""
inSortOrder	Text	→ ">" or "<"
inDelim	Text	→ List delimiter
Result	Text	← Modified list

Discussion

This method sorts the elements of a list and returns a modified copy.

If *inSortOrder* is ">", the elements are sorted in ascending order. If it is "<", they are sorted in descending order.

If *inDelim* is not passed, it defaults to "".

valueCount

version 3

valueCount(inList; inValue {; inDelim}) → Number

Parameter	Type	Description
inList	Text	→ Delimited list
inValue	<any>	→ Value to match
inDelim	Text	→ List delimiter
Result	Number	← Count of matching elements

Discussion

This method counts how many elements of *inList* match *inValue*, doing a case- and diacritical-sensitive match.

If *inDelim* is not passed, it defaults to ",".

valueCountNoCase

version 3

valueCountNoCase(inList; inValue {; inDelim}) → Number

Parameter	Type	Description
inList	Text	→ Delimited list
inValue	<any>	→ Value to match
inDelim	Text	→ List delimiter
Result	Number	← Count of matching elements

Discussion

This method counts how many elements of *inList* match *inValue*, doing a case- and diacritical-insensitive match.

If *inDelim* is not passed, it defaults to ",".

valueList

version 3

valueList(inField {; inDelim}) → Text

Parameter	Type	Description
inField	Field pointer	→ Field from which to get values
inDelim	Text	→ List delimiter
Result	Text	← New list

Discussion

This method returns a delimited list whose values are taken from the field pointed to by *inField*, one element for each record in the current selection of *inField's* table. *inField's*

type must be convertible to text with the **String** command (which includes Alpha and Text fields).

The selected record is not changed by this method.

If *inDelim* is not passed, it defaults to "".

a4d.utils

This library is composed of various utility methods that do *not* generate HTML for output to a page.

applyToSelection

version 4.5

applyToSelection(inTable; inStatement {; inTimeoutURL {; inTimeout}) → Boolean

Parameter	Type	Description
inTable	Table pointer	→ Table to delete records from
inStatement	Text	→ Code for APPLY TO SELECTION
inTimeoutURL	Text	→ URL of page to redirect to on timeout
inTimeout	Number	→ Seconds to wait until timeout
Result	Boolean	← True if successfully applied

Discussion

When using **APPLY TO SELECTION** in a multiuser database, it is of course important to ensure the entire selection is operated on, since some records may be locked.

This method repeatedly tries to call **APPLY TO SELECTION** (using **execute in 4d**) for the table pointed to by *inTable*, until either no locked records remain or *inTimeout* seconds have elapsed. If no locked records remain, the method returns *True*. If the timeout is reached, the method returns *False* and the set "LockedSet" will contain the locked records.

If *inTimeout* is not passed, it defaults to 2. If the timeout is reached and *inTimeoutURL* is passed and is not empty, a redirect is performed to that URL.

articleFor

version 3

articleFor(inNoun {; inBritishEnglish}) → Text

Parameter	Type	Description
inNoun	Text	→ Noun to get article for
inVowelH	Boolean	→ True to consider "h" a vowel sound
Result	Text	← "a" or "an"

Discussion

When generating messages in English, it is considered good form to use the proper article ("a" or "an") with noun phrases.

If you need to generate a message with dynamically generated noun phrases, this method will return the proper article. If *inVowelH* is *True*, "h" will be considered a vowel sound as well.

Example

```
method "whatAmI"($inKind)
  return ('You\'re `a4d.utils.articleFor($inKind)` $inKind')
end method

writebr(whatAmI("real peach")) // -> You're a real peach
writebr(whatAmI("idiot"))      // -> You're an idiot
```

blobToCollection

v5.0

blobToCollection(inBlob; ioOffset; inGlobal) → Collection

Parameter	Type		Description
inBlob	BLOB	→	BLOB data saved with <i>collectionToBlob</i>
ioOffset	Number	↔	Offset within inBlob to retrieve data
inGlobal	Boolean	→	<i>True</i> to create a global collection
Result	Collection	←	Deserialized collection

Discussion

This method recursively deserializes a collection serialized with **a4d.utils.collectionToBlob**. On entry, *ioOffset* should point to the byte offset within *inBlob* at which the serialized collection data is stored (typically zero). On exit, *ioOffset* points to the first byte beyond the serialized collection data.

If *inGlobal* is *True*, the collection returned is a global collection, otherwise it is a local collection.

blobToSession

v5.0

blobToSession(inBlob; ioOffset)

Parameter	Type		Description
inBlob	BLOB	→	BLOB data saved with <i>collectionToBlob</i>
ioOffset	Number	↔	Offset within inBlob to retrieve data

Discussion

This method is similar to the **blob to session** command, but it recursively deserializes a session serialized with **a4d.utils.sessionToBlob**. On entry, *ioOffset* should point to the

byte offset within *inBlob* at which the serialized session data is stored (typically zero). On exit, *ioOffset* points to the first byte beyond the serialized session data.

The deserialized session replaces the current session. For information on how this affects the current session, see “blob to session” on page 351.

camelCaseText

version 4.0

camelCaseText(inText {; inExceptions}) → Text

Parameter	Type	Description
inText	Text	→ Text to transform
inExceptions	Text	→ List of words to ignore
Result	Text	← Transformed text

Discussion

This method changes the case of the words in *inText* so that the first character of the word is uppercase and the remaining letters are lowercase. Words that appear in the space-separated list *inExceptions* are completely lowercased. If *inExceptions* is not passed it defaults to the word list “a at on in by”.

```
write(a4d.utils.camelCaseText("this is a test"))
// output -> This is a Test
```

chopText

version 4.0

chopText(inText) → Text

Parameter	Type	Description
inText	Text	→ Text to chop into chunks
Result	Text	← Chopped text

Discussion

This method chops *inText* into 80-character chunks separated by the string `""+""`. The primary purpose of this method is to prepare text for use with the command **execute in 4d**. When calling that command, the equivalent of a 4D **EXECUTE** is performed. Since 4D limits the total length of a single literal string to 80 characters, this method ensures that the entire text passed to **execute in 4d** is executed.

Example

```

$sql := 'select trans.tran_key,splits.split_key,trans.amount as
camount, splits.amount from trans,splits where trans.id =
splits.tran_id and split_key = \''$invnum\''
$cmd := 'pgsql Select(%d;"%s")' % ($connection; $sql)

$rowset:= execute in 4d(a4d.utils.chopText($cmd); *)

```

collectionToBlob**v5.0**

collectionToBlob(inCollection; ioBlob)

Parameter	Type	Description
inCollection	Collection	→ Collection to serialize
ioBlob	BLOB	↔ BLOB to serialize to

Discussion

This method recursively serializes a collection and appends the serialized data to *ioBlob*. Numeric items or elements of numeric arrays are checked for validity as collection handles, and if they are valid the collections they point to are serialized.

Example

```

$c := new collection("first"; "Tom"; "last"; "Bombadil")
$c{"company"} := new collection("name"; "Acme Corp.")
array longint($c{"company"}{"emps"}; 0)
$emp := new collection("name"; "Sam Gamgee"; "age"; 47)
$c{"company"}{"emps"} := $emp
$emp := new collection("name"; "Frodo Baggins"; "age"; 53)
$c{"company"}{"emps"} := $emp
c_blob($blob)
a4d.utils.collectionToBlob($c; $blob)
// save $blob somewhere

// in another script
$offset := 0
$c := a4d.utils.blobToCollection($blob; $offset; false)

```

Note: If you serialize with this method, be sure to deserialize with **a4d.utils.blobToCollection**.

cud

version 4.5

```
cud(cudAction; cudTable; cudProcessor {; attributes {; cudTimestampField {;
  cudTimestamp}}}) → Text
```

Parameter	Type	Description
cudAction	Text	→ Action to perform
cudTable	Table pointer	→ Table to operate on
cudProcessor	Text	→ Table-specific processing code
attributes	Collection	→ Values to pass to cudProcessor
cudTimestampField	Field pointer	→ Pointer to field holding a timestamp
cudTimestamp	Text	→ Current record's timestamp
Result	Text	← Status of action

Discussion

This method is a generalized processor for the most common operations on single records: creates, updates and deletes (hence the name “cud”). You supply a table-specific processing script and *cud* does the rest.

There are four actions you can pass in *cudAction*:

Action	Description
create	Create a new record
update	Update an existing record
update*	Update a record if it exists, create a new record if not
delete	Delete an existing record

cudTable is a pointer to the table on which you wish to perform the operation. *cud* takes care of setting and restoring the read write state of the table.

If *cudProcessor* begins with “/”, it is taken to be a web root-relative path to an executable script and is executed using **include**. Otherwise it is taken to be the name of an Active4D method (local or library) to execute.

attributes is a collection containing values you wish to pass on to *cudProcessor*. Under fusebox, typically this would be the *\$attributes* collection itself.

If you are using timestamps to perform optimistic locking (see “Using Timestamps with Optimistic Locking” on page 401), you can have *cud* do the timestamp checking when updating or deleting records. If *cudTimestampField* is non-nil and *cudTimestamp* is non-empty, the current value of the field pointed to by *cudTimestampField* will be compared to *cudTimestamp*.

cud returns a status which identifies the outcome of the operation. The built-in statuses returned by *cud* are:

You may also define your own status values in *cudProcessor*. The only status that *cud* really pays attention to is “success”; any status other than that is considered failure.

Status	Description
success	The operation completed successfully
missing	The record could not be found
conflict	Timestamp checking is being used and the timestamps do not match
locked	The record could not be unlocked

Phases

For each action, there are several possible *phases*: “query”, “update”, “delete”, “post” and “cleanup”. During each phase *cudProcessor* is executed, which gives you a chance to customize *cud*'s behavior accordingly. The details of each phase are covered below in the description of each action's flow of control.

Custom processors

Your custom processor needs to know the current state and return a status indicating success or failure.

In the case of an **include**-based *cudProcessor*, *cud* communicates its current state via local variables. In addition to the parameters passed to *cud* itself (which appear as local variables and should be considered read only), there are two additional variables:

- **\$cudPhase**: This is set to the current phase of the requested action, and should be considered read only. You use this variable to determine what to do in your processor.
- **\$cudResult**: This is set to the current status of the requested action, and may be changed by your processor. You may abort the action before the “post” phase by setting this variable to a value other than “success”.

In the case of a method-based *cudProcessor*, your method should be declared with the following signature:

```
method "myProcessor"($inAction; $inPhase; $inTable; $attributes;
$inTimestampField; $inTimestamp)
```

The method should return a text status. You may abort the action before the “post” phase by returning a value other than “success”.

If you want to maintain state in your *cudProcessor* between phases, it is recommended that you use the **_request** collection (see “_request” on page 330).

Now we have seen all of the elements that go into making *cud* work. At this point it will be useful to examine the flow of control of each action in more detail.

Create action

The “create” action has four phases, in the following order:

- **query**: Usually you will ignore this phase with the “create” action, since you don't need to find an existing record in *cudTable*. But there may be cases where, for example,

the creation of a record in *cusTable* is dependent on some value, either in a variable or in another table.

In such cases you would check for the dependent value, and if the test fails you would set the status to “missing”, or whatever error message you wish to use.

In this phase the read/write state of *cusTable* is whatever it was when *cus* was called (which is usually read only).

- **update:** If the status is still “success” after the “query” phase, the “update” phase is executed after **CREATE RECORD** is called and just before the newly created record is saved. In this phase *cusTable* will be read write and you must set *cusTable*’s fields to the appropriate values, typically retrieved from *\$attributes* in a fusebox application, or from **_query** or **_form** in non-fusebox applications.

If you want to use a transaction, you should start the transaction in this phase.

If you want to abort the record creation, set the status to something other than “success”.

- **post:** If the status is still “success” after the “update” phase, the “post” phase is executed just after the new created record is saved. In this phase you would typically perform operations on related tables that depend on the existence of a record in *cusTable*, or you may need to wait for *cusTable*’s trigger to execute before performing some postprocessing.
- **cleanup:** The “cleanup” phase is always executed, no matter what the status is. In this phase you perform any last-minute processing, for example committing or cancelling a transaction based on the status. If you are not using transactions, most of the time you will ignore this phase.

In this phase the read/write state of *cusTable* has already been restored to what it was before *cus* was called (usually read only).

Update action

The “update” action has the same four phases as the “create” action. The “post” and “cleanup” phases function identically to the “create” action. The “query” and “update” phases function as follows:

- **query:** In this phase you must perform a query (or some other operation) that will result in a selection of exactly one record in *cusTable*. If *cusTable* does not have exactly one record in its selection, *cus* will abort the action with a status of “missing”.
- **update:** If the status is still “success” after the “query” phase, the “update” phase is executed just before the record selected in the “query” phase is saved. Before this phase executes, *cus* has already set *cusTable* to read write and unlocked the record. In addition, if *cusTimestampField* was non-nil, *cus* has already verified that the record’s timestamp matches *cusTimestamp*.

In this phase you must set *cusTable*’s fields to the appropriate values, typically retrieved from *\$attributes* in a fusebox application, or from **_query** or **_form** in non-fusebox applications.

If you want to use a transaction, you should start the transaction in this phase.

If you want to abort the record update, set the status to something other than “success”.

Update* action

The “update*” action has the same four phases as the “update” action. The “post” and “cleanup” phases function identically to the “create” action. The flow of control during the “update*” action is as follows:

- 1 The “query” phase is executed as if the action were “update”.
- 2 If the “query” phase results in an empty selection in *cudTable*, *cudAction* is changed to “create” and then the action is restarted.
- 3 If the “query” phase results in a selection of one record in *cudTable*, execution continues as if the action were “update”.
- 4 If the “query” phase results in a selection of more than one record in *cudTable*, the action is aborted with a status of “missing”.

Delete action

The “delete” action has four phases like the “update” action. The “query”, “post” and “cleanup” phases function identically to the “update” action. Instead of an “update” phase, there is a “delete” phase which functions as follows:

- **delete:** If the status is still “success” after the “query” phase, the “delete” phase is executed just before the record selected in the “query” phase is deleted. Before this phase executes, *cud* has already set *cudTable* to read write and unlocked the record. In addition, if *cudTimestampField* was non-nil, *cud* has already verified that the record’s timestamp matches *cudTimestamp*.

In this phase you would typically update or delete related records. If there are no updates necessary outside of the main record deletion, you may ignore this phase.

If you want to use a transaction, you should start the transaction in this phase.

If you want to abort the record deletion, set the status to something other than “success”.

Examples

Let’s take a look at an example. In this example, we have an application that tracks companies and contacts. Both use an edit form which receives the record’s id in the query string. If a new record is being created, the id is “-1”.

Here is the HTML (quite simplified for the sake of this example) for the company edit form:

```
<form action="<%=fusebox.makeURL($XFA_onSubmit)%>" method="post">
<%
  // $rs is a RowSet populated in a query fuse
  $rs->first
  $row := $rs->getRow
  a4d.web.hideField("id"; $attributes{"id"})
  a4d.web.hideField("timestamp"; $row{"timestamp"})
%>
Name: <input name="name" type="text" value="<%= $row{"name"}%>" />
<br/>
Phone: <input name="phone" type="text"
          value="<%= $row{"phone"}%>" />
<br/>
<input name="b_save" type="submit" value="Save" />
<input name="b_cancel" type="submit" value="Cancel" />
<input name="b_delete" type="submit" value="Delete"
       style="margin-left: 70px" />
</form>
```

When the a button is clicked and the form is posted, fusebox routes the request to an action fuse:

```
if ($attributes{"b_cancel"} # "")
  // Either delete or update with create if not found
  $action := choose($attributes{"b_delete"}; \\  
                  "delete"; "update*")

  // Use a library method app.cudCompanies as processor,
  // and we want timestamp checking
  $result := a4d.utils.cud($action; ->[companies]; \\  
                      "app.cudCompanies"; $attributes; \\  
                      ->[companies]timestamp; $attributes{"timestamp"})

  if ($result = "success")
    redirect(fusebox.makeURL($XFA_onSuccess))
  else
    $query := build query string(""); "status"; $result)
    redirect(fusebox.makeURL($XFA_onFailure; $query))
  end if
else
  redirect(fusebox.makeURL($XFA_onCancel))
end if
```

That is all of the setup we need or *cud*. Now we take a look at our custom processor, the library method *app.cudCompanies*:

```
method "cudCompanies"($cudAction; $cudPhase; $cudTable; \\  
  $inStatus; $attributes; $cudTimestampField; $cudTimestamp)  
  
  case of  
    :($cudPhase = "query")  
      query([companies]; [companies]id = $attributes{"id"})  
  
    :($cudPhase = "update")  
      [companies]name := $attributes{"name"}  
      [companies]phone := $attributes{"phone"}  
  
    :($cudPhase = "delete")  
      // contacts for this company now belong to the  
      // special company "(None)" with id 0  
      if (not(a4d.utils.applyToSelection(->[contacts]; $code))  
        // oops, a contact record could not be unlocked  
        return ("locked")  
      end if  
  end case  
  
  return ("success")  
end method
```

As you can see, *cud* does most of the work and allows you to focus very tightly on the code that is specific to each table. Without *cud* you have to deal with the supporting logic, and you would end up copying, pasting and tweaking the same code for every table.

Note: Just so we are all clear on this principle: copy, paste and tweak is a *BAD THING*. Don't do it. Use *cud* instead.

deleteSelection

version 3

deleteSelection(inTable {; inTimeoutURL {; inTimeout}) → Boolean

Parameter	Type	Description
inTable	Table pointer	→ Table to delete records from
inTimeoutURL	Text	→ URL of page to redirect to on timeout
inTimeout	Number	→ Seconds to wait until timeout
Result	Boolean	← True if selection successfully deleted

Discussion

When deleting a selection of records in a multiuser database, it is of course important to ensure the entire selection is deleted, since some records may be locked.

This method repeatedly tries to delete the current selection in the table pointed to by *inTable*, until either no locked records remain or *inTimeout* seconds have elapsed. If no locked records remain, the method returns *True*. If the timeout is reached, the method returns *False* and the set “LockedSet” will contain the locked records.

If *inTimeout* is not passed, it defaults to 2. If the timeout is reached and *inTimeoutURL* is passed and is not empty, a redirect is performed to that URL.

filterCollection

version 4.0

filterCollection(inCollection; inFilter {; inGlobal}) → Collection

Parameter	Type	Description
inCollection	Collection Iterator	→ Collection to filter
inFilter	Text	→ Filter specification
inGlobal	Boolean	→ True to return a global collection
Result	Collection	← Filtered collection

Discussion

This method filters a collection and returns a new, filtered collection.

inFilter is matched against the keys of *inCollection* to determine which collection items are returned in the filtered collection. Each collection item whose key matches *inFilter* is copied to the filtered collection.

If the first character of *inFilter* is '+', the filter is an inclusion filter. If the first character is '-', the filter is an exclusion filter. If there is no '+/-', the filter defaults to an inclusion filter. If the first character after the optional '+/-' is '/', the rest of the filter is taken as a regular expression, and should follow the rules for regular expression match strings (see “regex match” on page 281). Otherwise the normal string matching rules (“@” is a wildcard) are in effect.

formatUSPhone

version 3

formatUSPhone(inPhone {; inFormat}) → Text

Parameter	Type	Description
inPhone	Text	→ Phone number to format
inFormat	Text	→ Format string for number
Result	Text	← Formatted phone number

Discussion

Given a phone number in any format, this method will do the following:

- If the number is empty, an empty string is returned.

- Any non-numeric characters are removed.
- If the remaining string is 11 digits and begins with “1”, the “1” is removed.
- If the remaining string is 10 digits it is formatted according to *inFormat*. If *inFormat* is not passed, the number is formatted as (000) 000-0000. Otherwise the number is formatted using the given format string using the % formatting operator. For example, to format the number as 000-000-000, pass a format of “%s-%s-%s”.
- If the string is not 10 digits, it is returned in its original form.

getMailMethod

version 4.0

getMailMethod → Text

Parameter	Type	Description
Result	Text	← Method to call for sending mail

Discussion

This method returns the name of the 4D method which is called by *a4d.utils.sendMail* to send a mail message. The default method is *A4D_SendMail*.

For more information, see “setMailMethod” on page 503.

getPictureDescriptor

version 3

getPictureDescriptor(inPicture) → Text

Parameter	Type	Description
inPicture	Picture	→ Picture to get info for
Result	Text	← Descriptive string

Discussion

This method returns a string which describes the width, height, and byte size of *inPicture*. This is primarily useful in debugging.

getPointerReferent

version 3

```
getPointerReferent(inPointer) → Text
```

Parameter	Type	Description
inPointer	Pointer	→ Pointer to get referent of
Result	Text	← Describes referent of inPointer

Discussion

This method returns a string which identifies the referent of *inPointer* in the same way it would appear in the 4D debugger. In other words, the output for different pointer types is:

Pointer type	Result
Table	->[table]
Field	->[table]field
Variable	->var
Array element	->array{index}

getSMTPAuthorization

version 4.0

```
getSMTPAuthorization(outUserName; outPassword)
```

Parameter	Type	Description
outUserName	Text	← SMTP account username
outPassword	Text	← SMTP account password

Discussion

This method returns the current username and password that will be used to authorize SMTP access when calling *sendMail*.

The username and password are set with *setSMTPAuthorization*.

getSMTPAuthPassword

version 4.0

getSMTPAuthPassword → Text

Parameter	Type	Description
Result	Text	← SMTP account password

Discussion

This method returns the current password that will be used to authorize SMTP access when calling *sendMail*.

The username and password are set with *setSMTPAuthorization*.

getSMTPAuthUser

version 4.0

getSMTPAuthUser → Text

Parameter	Type	Description
Result	Text	← SMTP account username

Discussion

This method returns the current username that will be used to authorize SMTP access when calling *sendMail*.

The username and password are set with *setSMTPAuthorization*.

getSMTPHost

version 4.0

getSMTPHost → Text

Parameter	Type	Description
Result	Text	← SMTP account username

Discussion

This method returns the current hostname that will be used for SMTP access when calling *sendMail*.

The SMTP host is set with *setSMTPHost*.

nextID**version 4.0**

nextID(inField {; inKeepSelection}) → Text

Parameter	Type	Description
inField	Field Pointer	→ ID field
inKeepSelection	Boolean	→ True to maintain selection/record
Result	Number	← Next highest ID

Discussion

Given a pointer to a field which contains a Longint, Alpha or Text ID, this method returns the number which is one higher than the maximum value in that field. If *inField* is an Alpha or Text field, it is converted to a number with the **Num** command.

If *inKeepSelection* is passed and is *True*, the current selection and record will be maintained. Otherwise they will be lost.

ordinalOf**version 3**

ordinalOf(inNumber) → Text

Parameter	Type	Description
inNumber	Number	→ Number to get ordinal for
Result	Text	← Number plus ordinal

Discussion

This method returns *inNumber* suffixed with the correct suffix to make it an ordinal number.

Example

```
write(a4d.utils.ordinalOf(1)) // -> 1st
write(a4d.utils.ordinalOf(10)) // -> 10th
```

parseConfig**version 4.5**

parseConfig(inPathOrDocRef {; inEOL}) → Collection

Parameter	Type	Description
inPathOrDocRef	Text/Time	→ Path to file or document reference
inEOL	Text	→ Line ending

Discussion

This method parses a text file with one or more lines in the form <key>=<value> into a collection.

If *inPathOrDocRef* is of type *Time*, it must be a valid document reference to an open file that was opened with the `openFile` command. In this case you are responsible for closing the file after calling this method.

Otherwise *inPathOrDocRef* should be a text path to the config file you want to parse. The path should follow the rules or paths used with the `openFile` command. If the file is successfully opened, it is automatically closed before this method returns. If the file cannot be opened, an empty collection is returned.

If *inEOL* is not passed, the line endings are assumed to be LF (“\n”). If your config file has line endings other than LF, you must pass the line ending in *inEOL*. The last line of the config file does not need to be terminated with a line ending.

Blank lines and white space around the keys and values are discarded. In addition, if the first characters of a line are “” (4D comment character) or “/” (Active4D line comment characters), the line is ignored. This allows you to comment the config file.

Examples

Let’s assume a site we are developing has to send emails. During testing, we do not want the emails to go to the intended recipients, we want them to come to us. In addition, the SMTP host on the deployment server is different than the one we use during testing.

Instead of hard coding hostnames and email addresses into our code, we can use a config file. We decide to put a file called “mail.ini” in the database directory. Here’s what our config file might look like:

```
// Use this file to configure the settings for your mail server.
// If you want to force mail to go to a certain address, enter
// the "to" address here. To use the normal destination
// address, leave the "to" value empty.
// If a "to" address is entered here, the mail message
// will display the original "to" address.

host = smtp.foobar.com
to   = foo@bar.com
```

To ensure our config file is accessible, we set the “safe doc dirs” option in Active4D.ini:

```
safe doc dirs = <default>
```

In the “On Application Start” method in the Active4D library, we add this code:

```
$path := join paths(default directory(*); "mail.ini")
$config := a4d.utils.parseConfig($path)
a4d.utils.setSMTPHost($config{"host"})
globals{"mail.to"} := $config{"to"}
```

Then, to build our mail message we do this:

```
$testing := ""
$from := "staff@mycompany.com"
$to := globals{"mail.to"}

if ($to)
    $testing += 'Original To: [employee]email\n'
else
    $to := [employee]email
end if

if ($testing)
    $testing := "[This is a test message]\n" + $testing + "\n"
end if

$name := [employee]first_name + " " + [employee]last_name
$msg := "|System message\n%sHello,\n\nOn %s, " + \\  

    "%s foode the bar.\n\nSincerely,\n\n    - The System"
$msg := $msg % ($testing; string(current date); $name)
$err := a4d.utils.sendMail($msg; $from; $to)
```

reverseArray

version 3

```
reverseArray(ioArray)
```

Parameter	Type	Description
ioArray	Array	↔ Array to reverse

Discussion

This method reverses the elements of *ioArray* in place.

selectionRangeToCollection

v6.0r7

```
selectionRangeToCollection(inStart; inEnd; inFields {; inWantGlobal }) → Collection
```

Parameter	Type	Description
inStart	Number	→ Starting index in selection
inEnd	Number	→ Ending index in selection
inFields	Array/Text/Pointer	→ Field specification
inWantGlobal	Boolean	→ True to return a global collection

Discussion

This method is the same as **selectionToCollection**, but allows you to specify a range of selected records as you would with **SELECTION RANGE TO ARRAY**.

selectionToCollection

v6.0r7

```
selectionToCollection(inFields {; inWantGlobal }) → Collection
```

Parameter	Type	Description
inFields	Array/Text/Pointer	→ Field specification
inWantGlobal	Boolean	→ True to return a global collection

Discussion

This method is analogous to **SELECTION TO ARRAY**, but instead of copying the field data into a number of different array variables, the data is copied into arrays within a single collection.

The resulting collection has one item per field, each of which contains an array of values for that field. Like **SELECTION TO ARRAY**, the table of the first field is considered the main table.

inFields may be either:

- A semicolon-delimited list of [table]field names and optional aliases
- An array of [table]field names and optional aliases
- An array of field pointers
- A table pointer (which must be passed in a variable)

When passing a list or array of [table]field names, leading and trailing whitespace is ignored. If a [table]field is followed by an item which does not begin with "[", the next item is used as the key for that field in the collection. This allows you to map database field names to something else.

If you pass an array of field pointers, it is the same as passing an array of [table]field names for those fields.

If you pass a table pointer, it is the same as passing an array of all [table]field names for that table.

If a field name is not aliased, the field name is used as the collection key for fields belonging to the main table. For unaliased fields in other tables, the collection key will be *table.field*.

Example

In this example, we specify the list of fields and aliases. Because leading and trailing whitespace is ignored, we can use a multi-line heredoc string:

```
$fields := ""
[contacts]id;
[contacts]first_name;first;
[contacts]last_name;last;
[company]id;
[company]name""
$collection := a4d.utils.selectionToCollection($fields)

// $collection has 5 array items:
// id, first, last, company.id, company.name
```

sendMail

version 3
modified version 4.0

sendMail(inMailFileOrText; inFrom; inTo; inCC; inBCC; inAttachmentPath; inUserData; inHeaders) → Number

Parameter	Type	Description
inMailFileOrText	Text	→ Path to a file or text to send/execute
inFrom	Text	→ From address
inTo	Text	→ To addresses
inCc	Text	→ Cc addresses
inBcc	Text	→ Bcc addresses
inAttachmentPath	Text	→ Unix path to attachment
inUserData	Collection	→ Data for included file
inHeaders	Text	→ Extra headers to add to message
Result	Number	← Error code, 0 means success

Discussion

This method is a front end to the 4D method *A4D_SendMail* that makes it easy to dynamically generate email messages.

If the first character of *inMailFile* is not '*!*', it is considered a path to a file to include, relative to the file that called this method. Because the file is included within the context of this method, the caller's local variables are not available. To access the caller's local variables, put the values you want to access in a collection and pass that collection in *inUserData*.

The output of the include becomes the source of the email message.

If the first character of *inMailFile* is '*!*', the text following it is executed using the Active4D **execute** command. Any output from that execution that would normally go to the response buffer is saved and becomes the source of the email message.

If the first character of *inMailFile* is '*|*', the text following it becomes the source of the email message.

The first line of the email source is taken as the subject of the message, and must be separated from the body of the message by a linefeed ("*\n*"). Thus you must save an include file with Unix line endings when you edit it.

The rest of the email source becomes the body of the message.

inTo, *inCc*, and *inBcc* can all contain more than one address, separated by commas.

To add an attachment, pass the full Unix-style path to the attachment in *inAttachmentPath*.

If *inHeaders* is passed, it should be a linefeed-delimited list of header:value pairs.

Note: Before using this command you must call `setSMTPHost`, and `setSMTPAuthorization` if the SMTP host requires account authorization. Also, the subject *must* be separated from the body by a linefeed ("`\n`"), not a carriage return ("`\r`").

Example

When a new user logs into the system, we want to send a welcome email with their user information. Here is the email include file called "welcome.inc":

```
Welcome to ACME Corporation!
Dear <%= $inUserData{"firstname"}%>,

Welcome to the growing family of ACME Corporation members. Your
login information is as follows:

username: <%= $inUserData{"username"}%>
password: <%= $inUserData{"password"}%>

To login, please go to www.acme.com/login.

Regards,

ACME Corporation
```

To generate the email, we first do a query that generates a `RowSet` with the columns "email", "username", "password", and "firstname". We then pass this to `sendMail`:

```
$map := ""
email: [users]email;firstname:[users]firstname;
username:[users]username;password:[users]password""

$rs := RowSet.NewFromSelection(->[users]; $map)
$rs->first
$row := $rs->getRow
a4d.utils.sendMail("email/welcome.inc"; "sales@acme.com"; \\
  $row{"email"}; ""; ""; ""; $row; \\
  "Reply-To: noreply@acme.com")
```

sessionToBlob

v5.0

`sessionToBlob(ioBlob)`

Parameter	Type	Description
ioBlob	BLOB	↔ BLOB to serialize to

Discussion

This method recursively serializes the current session and appends the serialized data to *ioBlob*. Numeric items or elements of numeric arrays are checked for validity as collection handles, and if they are valid the collections they point to are serialized.

Note: If you serialize with this method, be sure to deserialize with **a4d.utils.blobToSession**.

setMailMethod

version 4.0

`setMailMethod(inMethod)`

Parameter	Type	Description
inMethod	Text	→ 4D method name

Discussion

This method sets the name of the 4D method which is called by *a4d.utils.sendMail* to send a mail message. The default method is *A4D_SendMail*.

If you specify a method other than *A4D_SendMail*, its parameters must be identical in position and type to *A4D_SendMail*.

A4D_SendMail uses 4D Internet Commands to send email immediately. If you want to use some other method to send email, for example if you want to queue the mail for later delivery or use some other plugin to send mail, you can use *setMailMethod* to call your own custom method.

setSMTPAuthorization

version 4.0

```
setSMTPAuthorization(inUserName; inPassword)
```

Parameter	Type	Description
inUserName	Text	→ SMTP account username
inPassword	Text	→ SMTP account password

Discussion

This method sets the username and password that will be used to authorize SMTP access when calling *sendMail*.

setSMTPHost

version 4.0

```
setSMTPHost(inHost)
```

Parameter	Type	Description
inHost	Text	→ SMTP hostname

Discussion

This method sets the hostname that will be used for SMTP access when calling *sendMail*.

truncateText

version 4.0

```
truncateText(inText; inWidth; inTruncateMode) → Text
```

Parameter	Type	Description
inText	Text	→ Text to truncate
inWidth	Number	→ Width to truncate to
inTruncateMode	Number	→ How to truncate
Result	Text	← Truncated text

Discussion

This method truncates *inText* to the character width specified by *inWidth*. If **Length**(*inText*) < 3 or **Length**(*inText*) < *inWidth*, *inText* is returned as is. Otherwise **Length**(*inText*) - *inWidth* characters are removed from *inText* and are replaced by an ellipsis ("...").

The placement of the ellipsis is controlled by *inTruncateMode*. There are three modes which are defined as library constants:

Truncate modes

kTruncateText_Start

kTruncateText_Middle

kTruncateText_End

Example

If you know you will have a very long string, such as a file path, you may want to display only part of it onscreen, like so:

```
$path := "/Home/Users/Spock/Documents/biz/webapps/contacts/docs"
write(a4d.utils.truncateText($path; 20; \\
    a4d.utils.kTruncateText_Start))
// -> ...ebapps/contacts/docs
write(a4d.utils.truncateText($path; 20; \\
    a4d.utils.kTruncateText_Middle))
// -> /Home/User...tacts/docs
write(a4d.utils.truncateText($path; 20; \\
    a4d.utils.kTruncateText_End))
// -> /Home/Users/Spock/Do...
```

unlockAndLoad**version 3**
modified version 4.0

unlockAndLoad(inTable {; inRecNum {; inTimeoutURL {; inTimeout}}}) → Boolean

Parameter	Type	Description
inTable	Table pointer	→ Table to load record from
inRecNum	Number	→ Record number of record to unlock, or -1 to use current record
inTimeoutURL	Text	→ URL of page to redirect to if timeout
inTimeout	Number	→ Seconds to wait before timing out
Result	Boolean	← True if record successfully unlocked

Discussion

This method tries repeatedly to load an unlocked copy of the record with the record number *inRecNum* in the table pointed to by *inTable*, until either the record becomes unlocked or *inTimeout* seconds elapses.

Basically, you should be using this method any time you need to update a record.

If *inRecNum* is not passed or is -1, the current record in the table pointed to by *inTable* is used. If *inTimeout* is not passed, by default *unlockAndLoad* will wait 2 seconds before timing out. If *inTimeoutURL* is passed in and is non-empty, after a timeout a redirect will be performed to that URL.

If the record is successfully unlocked, *True* is returned and the table pointed to by *inTable* is left in read write mode. If a timeout occurs while waiting for a lock, the read only state of the table is restored to what it was before the method was called, and if *inTimeoutURL* is empty *False* is returned.

Example

A typical scenario is the posting of an edit form for a record. In the form processing script you need to update the record. Typically you would do something like this:

```
query([contacts]; [contacts]id = $attributes{"cid"})

if (records in selection([contacts]) = 1)
  if (a4d.utils.unlockAndLoad(->[contacts]))
    [contacts]email := $attributes{"f_email"}
    [contacts]phone := $attributes{"f_phone"}
    save record([contacts])
    unload record([contacts])
    read only([contacts])
  end if
end if
```

validPrice

version 3
modified version 4.0

validPrice(inPrice; inCurrencyMark; inDecimalSep) → Boolean

Parameter	Type	Description
inPrice	Number/Text	→ Price to check
inCurrencyMark	Text	→ Optional currency mark to allow, defaults to "\$"
inDecimalSep	Text	→ Decimal separator character, defaults to "."
Result	Boolean	← True if inPrice fits the pattern

Discussion

This method validates the format of a price. If *inPrice* is a number, it is converted to a string with two decimal places and the decimal separator *inDecimalSep*.

yearMonthDay

version 3
modified version 4.0

yearMonthDay({inShortDate {; inDate}}) → Text

Parameter	Type		Description
inShortDate	Boolean	→	True to use 2-digit year, defaults to False
inDate	Date	→	Date to format, defaults to current date
Result	Text	←	Formatted date

Discussion

This method converts *inDate* to the format YYYYMMDD if *inShortDate* is *False* or is not passed.

If *inShortDate* is passed and is *True*, *inDate* is converted to the format YYMMDD.

If *inDate* is not passed or is !00/00/00!, it defaults to the current server date.

a4d.web

This library is composed of methods that are primarily designed to make it easier to work with forms, query strings, and SELECT lists.

Note: The dump <something> methods have been moved to a separate a4d.debug library.

br**v6.0**

br

Discussion

This method returns a
 tag if the “doctype” option in Active4D.ini is “html”, or
 if the doctype is “xhtml”.

buildOptionsFromArrays**version 3**

```
buildOptionsFromArrays(inOptions; inValues; inInitialIndex {; inInitialValue})
```

Parameter	Type	Description
inOptions	Array	→ Option texts
inValues	Array	→ Option values
inInitialIndex	Number	→ Index of option to select, 0 to use inInitialValue
inInitialValue	<any>	→ Value of option to select

Discussion

This method writes a series of HTML *OPTIONS* to the output buffer, one for each element in the *inOptions* array. The text of each *OPTION* comes from *inOptions*, and the value comes from *inValues*.

Both *inOptions* and *inValues* can be any type to which the **String** command can be applied to its elements (which includes String/Text Arrays). If *inSelect* is passed in, it must be assignment compatible with *inValues*. If *inInitialIndex* > 0, it is used as the starting index and *inInitialValue* is ignored.

Note that this method only builds the *OPTIONS* of a *SELECT* object, not the *SELECT* object itself.

Note: This method was previously called `buildArrayValueList`, which will continue to work in this release but is now deprecated.

Example

Given arrays of country names and codes that you have cached in the `globals` collection, you could build a select list that would default to United States as the country and would return the country code like this:

```
<select name="f_country" id="f_country" size="1">
<% a4d.web.buildOptionsFromArrays(globals{"country_names"}; \\  
    globals{"country_codes"}; 0; "US") %>
</select>
```

buildOptionsFromLists

version 3

`buildOptionsFromLists(inOptions; inValues; inInitialIndex {; inInitialValue {; inDelimiter})`

Parameter	Type	Description
<code>inOptions</code>	Text	→ Delimited list of option texts
<code>inValues</code>	Text	→ Delimited list of option values
<code>inInitialIndex</code>	Number	→ Index of option to select, 0 to use <code>inInitialValue</code>
<code>inInitialValue</code>	Text	→ Value of option to select
<code>inDelimiter</code>	Text	→ List delimiter, defaults to “;”

Discussion

This method writes a series of HTML *OPTIONS* to the output buffer, one for each element in the *inOptions* list. The text of each *OPTION* comes from *inOptions*, and the value comes from *inValues*.

If *inInitialIndex* > 0, it is used as the starting index and *inInitialValue* is ignored.

Note that this method only builds the *OPTIONS* of a *SELECT* object, not the *SELECT* object itself.

Note: This method was previously called `buildSelectValueMenu`, which will continue to work in this release but is now deprecated.

Example

To build a choice list of US states and select the first state by default, you could do the following:

```
<select name="f_state" id="f_state" size="1">
<% a4d.web.buildOptionsFromLists( \\  
    a4d.web.kUS_SortedStateNames; \\  
    a4d.web.kUS_UnsortedStateCodes; 1) %>
</select>
```

buildOptionsFromOptionArray

version 3

`buildOptionsFromOptionArray(inOptions; inInitialIndex {; inInitialValue})`

Parameter	Type	Description
<code>inOptions</code>	Array	→ Contains option texts
<code>inInitialIndex</code>	Number	→ Index of option to select, 0 to use <code>inInitialValue</code>
<code>inInitialValue</code>	<any>	→ Value of option to select

Discussion

This method writes a series of HTML *OPTIONS* to the output buffer, one for each element in the *inOptions* array. The text of each *OPTION* comes from *inOptions*. If *inInitialIndex* > 0, the value of each *OPTION* is its index. If *inInitialIndex* > 0, the value of each *OPTION* comes from *inOptions*.

inOptions can be any type to which the **String** command can be applied to its elements (which includes String/Text Arrays). If *inInitialValue* is passed in, it must be assignment compatible with *inOptions*. If *inInitialIndex* > 0, it is used as the starting index and *inInitialValue* is ignored.

Note that this method only builds the *OPTIONS* of a *SELECT* object, not the *SELECT* object itself.

Note: This method was previously called `buildArrayList`, which will continue to work in this release but is now deprecated.

Example

To build a choice list of cached status codes, you could do this:

```
<select name="f_status" id="f_status" size="1">
<% a4d.web.buildOptionsFromOptionArray( \\  
    globals{"customer.status"}; 0; $row{"status"}) %>
</select>
```

buildOptionsFromOptionList

version 3

`buildOptionsFromOptionList(inOptions; inInitialIndex {; inInitialValue})`

Parameter	Type	Description
<code>inOptions</code>	Text	→ Delimited list of option texts
<code>inInitialIndex</code>	Number	→ Index of option to select, 0 to use <code>inInitialValue</code>
<code>inInitialValue</code>	Text	→ Value of option to select

Discussion

This method writes a series of HTML *OPTIONS* to the output buffer, one for each element in the *inOptions* list. The text of each *OPTION* comes from *inOptions*. If *inInitialIndex* > 0, the value of each *OPTION* is its index. If *inInitialIndex* > 0, the value of each *OPTION* comes from *inOptions*.

If *inInitialIndex* > 0, it is used as the starting index and *inInitialValue* is ignored.

Note that this method only builds the *OPTIONS* of a *SELECT* object, not the *SELECT* object itself.

Note: This method was previously called `buildSelectMenu`, which will continue to work in this release but is now deprecated.

Example

To build a choice list of US state codes that defaults to the first state, you could do this:

```
<select name="f_state" id="f_state" size="1">
<% a4d.web.buildOptionsFromOptionArray( \\  
    a4d.web.kUS_State; 1) %>
</select>
```

buildOptionsFromRowSet**version 4.0**

```
buildOptionsFromRowSet(inRowSet; inOptionColumn; inValueColumn; inInitialIndex
{; inInitialValue})
```

Parameter	Type	Description
inRowSet	RowSet	→ Rows from which to get data
inOptionColumn	Text	→ Name of column which contains option texts
inValueColumn	Text	→ Name of column which contains option values
inInitialIndex	Number	→ Index of option to select, 0 to use inInitialValue
inInitialValue	Text	→ Value of option to select

Discussion

This method writes a series of HTML *OPTIONS* to the output buffer, one for each row in *inRowSet*. The text of each *OPTION* comes from the column *inOptionColumn*. The value of each *OPTION* comes from the column *inValueColumn* if *inValueColumn* is not an empty string, or from the column *inOptionColumn* otherwise.

If *inInitialIndex* > 0, it is used as the starting index and *inInitialValue* is ignored.

Note that this method only builds the *OPTIONS* of a *SELECT* object, not the *SELECT* object itself.

Example

To build a multiple choice list of sport names and codes that defaults to nothing selected, you could do this:

```
<select name="f_sports" id="f_sports" size="10"
multiple="multiple">
<%
a4d.web.buildOptionsFromRowSet($rsSports; "name"; "code"; -1)
%>
</select>
```

buildOptionsFromSelection

version 3

```
buildOptionsFromSelection(inOptionField; inValueField; inInitialValue)
```

Parameter	Type	Description
inOptionField	Field pointer	→ Points to option texts
inValueField	Field pointer	→ Points to option values
inInitialValue	<any>	→ Value of option to select initially

Discussion

This method writes a series of HTML *OPTION*s to the output buffer, one for each record in the current selection of *inValueField*'s table. The text of each *OPTION* comes from *inOptionField*. The value of each *OPTION* comes from *inValueField*.

If **String**(*inValueField*->) evaluates to "0", the option text is set to "<Default>".

Note that this method only builds the *OPTION*s of a *SELECT* object, not the *SELECT* object itself.

Note: This method was previously called `buildRecordList`, which will continue to work in this release but is now deprecated. Also note that `buildRecordList` reversed the order of `inOptionField` and `inValueField`.

Example

To build a choice list of vendor names and IDs that defaults to the first one selected, you could do this:

```
<select name="f_vendors" id="f_vendors" size="1">
<% a4d.web.buildOptionsFromSelection( \\  
    ->[vendors]name; ->[vendors]id; [vendors]id) %>
</select>
```

checkSession

version 3

```
checkSession(inItemToCheck; inRedirectURL {; inQueryString})
```

Parameter	Type	Description
inItemToCheck	Text	→ Name of item to check for in session
inRedirectURL	Text	→ URL of page to redirect to
inQueryString	Text	→ Query string to pass on redirect

Discussion

It is important that you detect when a session has timed out or when no session exists. This is easily done by designating one session item (such as a user id) which only exists when a user has successfully logged in to the site.

You then need to check the session before displaying every page that relies on a current session. This method is a convenience method for checking the session. You pass the designated session item in *inItemToCheck*. If the item does not exist, a redirect is performed to *inRedirectURL*, passing *inQueryString*.

Note: If you are using Fusebox, do not use this method.

Example

```
<% checkSession("user.id"; "/timeout.a4d") %>
```

checkboxState

version 3

```
checkboxState(inState)
```

Parameter	Type	Description
inState	Boolean Text	→ State of a checkbox

Discussion

This method makes it easier to set dynamic checkboxes in your pages. If *inState* is a Boolean and *True*, or if it is text and is "1", the text "checked="checked"" is written to the output buffer.

Example

Suppose you have a series of checkboxes on a form that reflect data from a RowSet. You could set them like this:

```
<input name="f_dorm" id="f_dorm" type="checkbox"
  <%checkboxState($row{"dorm"})%>> Will live in dorm
```

collectionItemsToQuery

version 3

collectionItemsToQuery(inCollection; inItems {; inSwitches}) → Text

Parameter	Type	Description
inCollection	Collection Iterator	→ Collection from which to get items
inItems	Text	→ Semicolon-delimited list of item keys
inSwitches	Text	→ Switches to control query formatting
Result	Text	← Query string

Discussion

This method constructs a query string (without a leading "?") from the items in *inCollection* whose keys match the keys in the semicolon-delimited list *inItems*. White space around the key names in *inItems* is automatically trimmed.

inSwitches determines the formatting of the query string, and follows the same format as the switches passed to **build query string**. For more information, see "build query string" on page 275.

Example

When you are displaying a page that is based on some contextual information derived from a posted form or the query string, you usually need to maintain that contextual information in any links which exit the page.

For example, consider the following typical "drill-drown" scenario:

- The user selects a company
- From a list of contacts in the company, a contact is clicked on to view his or her details
- From a list of phone calls, one call is clicked on to view the details of the call

At each step along the way, you want to know the full context so you know how the user got to where they are and how they can get back. So you need to carry the context information (such as company ID / contact ID / call ID) forward as the user drills down. This method makes it easy to construct links with query strings that contain all the context information you need.

In the company/contacts/phone calls example cited above, the links to display a call detail might look like this:

```
<%
  // We are using Fusebox, so the query string for this
  // page is in $attributes
  // cid = company id, ctid = contact id
  $query := a4d.web.collectionItemsToQuery($attributes; \\
    "cid;ctid")
%>
<a href="<%=fusebox.makeLink($XFA_onViewCallDetail; $query)%>"
  title="View call detail">$row{"number"}</a>
```

collectionToQuery

version 4.0

collectionToQuery(inCollection {; inFilter}) → Text

Parameter	Type	Description
inCollection	Collection Iterator	→ Collection from which to get items
inFilter	Text	→ Text to match against
Result	Text	← Query string

Discussion

This method is identical to `embedCollection`, but instead of creating hidden form fields, constructs a query string (without a leading "?") from the items in `inCollection` whose keys match the filter specified by `inFilter`.

For more information on how the filtering works, see “`embedCollection`” on page 517.

embedCollection

version 3

embedCollection(inCollection {; inFilter})

Parameter	Type	Description
inCollection	Collection Iterator	→ Collection from which to get items
inFilter	Text	→ Text to match against

Discussion

When you are displaying a form that is based on some contextual information derived from a posted form or the query string, you usually need to maintain that contextual information in hidden fields within the form, so that when the form is posted you can determine the context.

For example, consider the following typical scenario:

- The user selects a company
- From a list of contacts in the company, a contact is clicked on to view his or her details

At each step along the way, you want to know the full context so you know how the user got to where they are and how they can get back. So you need to carry the context information (such as company ID / contact ID) forward as the user navigates through the data.

If all of the context information is in a single collection, and the keys of the collection items can be included or excluded by a pattern, this method makes it easy to embed all of the context information you need within a form.

Note: If the keys of the items you want to embed do not follow a pattern (which is often the case), use **embedCollectionItems** instead of this method.

inFilter is matched against the keys of *inCollection* to determine which collection items are embedded in the form. For each collection item whose key matches *inFilter*, a hidden form field is created whose id/name is the item key and whose value is the item value. If a collection item is an array, a hidden form field will be created for each element of the array.

If the first character of *inFilter* is '+', the filter is an inclusion filter. If the first character is '-', the filter is an exclusion filter. If there is no '+/-', the filter defaults to an inclusion filter. If the first character after the optional '+/-' is '/', the rest of the filter is taken as a regular expression, and should follow the rules for regular expression match strings (see "regex match" on page 281). Otherwise the normal string matching rules ('@' is a wildcard) are in effect.

Example

In the company/contacts example cited above, the edit form for the contact might look like this:

```
<form id="form" name="form"
  action="<%=fusebox.makeURL($XFA_onSubmit)%>"
  method="post"
>
<div>
<input id="f_name" name="f_name" type="text"
  value="<%= $row{"name"}%>" /><br />
<input id="f_phone" name="f_phone" type="text"
  value="<%= $row{"phone"}%>" /><br />
<input id="x_submit" name="x_submit" type="submit" value="Save"
/>
<%
  // embed the context
  a4d.web.embedCollection($attributes; "f_@")
%>
</div>
</form>
```

embedCollectionItems

version 3

```
embedCollectionItems(inCollection; inItems)
```

Parameter	Type	Description
inCollection	Collection Iterator	→ Collection from which to get items
inItems	Text	→ Semicolon-delimited list of item keys

Discussion

This method serves the same purpose as *collectionItemsToQuery*, but instead of returning a query string, embeds the items as hidden form items. For more details see the discussion for *collectionItemsToQuery*.

embedFormVariableList

version 3

```
embedFormVariableList(inItems)
```

Parameter	Type	Description
inItems	Text	→ Semicolon-delimited list of form variables

Discussion

This method is equivalent to:

```
embedCollectionItems(_form; $inItems)
```

For more information, see the discussion of “embedCollectionItems” on page 519.

embedFormVariables

version 3

```
embedFormVariables({inFilter})
```

Parameter	Type	Description
inFilter	Text	→ Text to match against

Discussion

This method is equivalent to:

```
embedCollection(_form; $inFilter)
```

For more information, see the discussion of “embedCollection” on page 517.

embedQueryParams

version 3

embedQueryParams({inFilter})

Parameter	Type	Description
inFilter	Text	→ Text to match against

Discussion

This method is equivalent to:

```
embedCollection(_query; $inFilter)
```

For more information, see the discussion of “embedCollection” on page 517.

embedVariables

version 3

embedVariables({inFilter})

Parameter	Type	Description
inFilter	Text	→ Text to match against

Discussion

This method is equivalent to:

```
embedCollection(getVariablesIterator; $inFilter)
```

For more information, see the discussion of “embedCollection” on page 517 and “getVariablesIterator” on page 522.

emptyTag

v6.0

```
emptyTag(inTag {; inAttributes})
```

Parameter	Type	Description
inTag	Text	→ Tag name
inAttributes	Text	→ Tag attributes

Discussion

This method returns *inTag* and *inAttributes* wrapped correctly according to the doctype set in the “doctype” option of Active4D.ini and according to which tags can actually be empty in XHTML.

This is useful if you are writing generic code that might run under any doctype. If you know the doctype of your site, this method is not necessary.

Examples

```
write(a4d.web.emptyTag("hr"; "style=\"height:3px\""))
// html: <hr style="height:3px">
// xhtml: <hr style="height:3px" />

write(a4d.web.emptyTag("div"; "id=\"foo\""))
// html: <div id="foo"></div>
// xhtml: <div id="foo"></div>
```

formVariableListToQuery

version 3

```
formVariableListToQuery(inVariables {; inSwitches})
```

Parameter	Type	Description
inVariables	Text	→ Semicolon-delimited list of form variables
inSwitches	Text	→ Controls the formatting of the query

Discussion

This method is equivalent to:

```
collectionItemsToQuery(_form; $inVariables; $inSwitches)
```

If *inSwitches* is not passed, it defaults to an empty string. For more information, see the discussion of “collectionItemsToQuery” on page 516.

getEmptyFields

version 3

getEmptyFields(inFieldList) → Text

Parameter	Type	Description
inFieldList	Text	→ Semicolon-delimited list of form fields to validate
Result	Text	← Semicolon-delimited list of empty fields

Discussion

This method checks the *form variables* collection for each field named in the list *inFieldList*. If the field does not exist or is empty, it is added to the result list.

getUniqueID

version 3

getUniqueID → Text

Parameter	Type	Description
Result	Text	← Per-request unique identifier

Discussion

Sometimes it is necessary to force (lousy) browsers or proxy caches into thinking that the same page is a new page in order to prevent the page from being cached. The simplest way to do this is to append a unique query parameter to the query string of the URL. This forces clients to think it is a different page.

This method will give you a per-request unique identifier that you can put into a query string.

getVariablesIterator

version 3

getVariablesIterator → Iterator

Parameter	Type	Description
Result	Iterator	← Either <code>_form</code> or <code>_query</code>

Discussion

This method returns an iterator for the collection in which values were passed to the current page. In other words, whether the current page was reached via a link, a form posted with the “get” method, or a form posted with the “post” method, this method will

return an iterator that gives you access to the query string or form variables appropriately.

hideField

version 3

hideField(inName; inValue {; inEncode})

Parameter	Type	Description
inName	Text	→ Name/id of hidden field
inValue	<any>	→ Value of hidden field
inEncode	Boolean	→ True to HTML encode inValue

Discussion

This method writes an XHTML-compatible hidden form field to the output buffer. *inValue* must be convertible by the **String** command (which includes strings/text).

If *inEncode* is not passed or is passed and is *True*, *inValue* is HTML encoded.

hideUniqueField

version 3

hideUniqueField(inName)

Parameter	Type	Description
inName	Text	→ Name/id of hidden field

Discussion

This method writes an XHTML-compatible hidden form field to the output buffer with a per-request unique value. For more information see “getUniqueID” on page 522.

saveFormToSession

version 3

saveFormToSession({inFilter})

Parameter	Type	Description
inFilter	Text	→ Text to match against

Discussion

If you want to save one or more form variables to the session, and the form variable names can be included or excluded by a pattern, this method makes it easy to do so.

inFilter is matched against the names of the form variables to determine which ones are saved in the session. For each form variable whose name matches *inFilter*, a session item is created whose key is the form variable name and whose value is the form variable value.

If the first character of *inFilter* is '+', the filter is an inclusion filter. If the first character is '-', the filter is an exclusion filter. If there is no '+/-' the filter defaults to an inclusion filter. If the first character after the optional '+/-' is '/', the rest of the filter is taken as a regular expression, and should follow the rules for regular expression match strings (see "regex match" on page 281). Otherwise the normal string matching rules ('@' is a wildcard) are in effect.

validateTextFields

version 3

validateTextFields(inFieldList) → Text

Parameter	Type	Description
inFieldList	Text	→ Semicolon-delimited list of form fields to validate
Result	Text	← Semicolon-delimited list of empty fields

Discussion

This method is an alias for **getEmptyFields**.

Note: This method is deprecated. Please use **getEmptyFields** instead.

validEmailAddress

version 3
modified version 4.0

validEmailAddress(inAddress) → Boolean

Parameter	Type	Description
inAddress	Text	→ Email address to validate
Result	Boolean	← True if valid

Discussion

This method validates that *inAddress* is valid using a regular expression that should cover the vast majority of possible email addresses.

Note: Full validation according to RFC822 would require a regular expression that is 6K characters in length!

warnInvalidField

version 3

```
warnInvalidField(inCurrentField; inInvalidField; inWarningURL)
```

Parameter	Type	Description
inCurrentField	Text	→ The current field being checked
inInvalidFields	Text	→ Semicolon-delimited list of invalid form fields

Discussion

\$inCurrentField->TextThe current field we are checking

\$inInvalidFields->TextSemicolon-delimited list of invalid form fields

\$inWarningURL->TextURL of warning icon to display

If \$inCurrentField is in the list \$inInvalidFields, an image tag is written to display \$inWarningURL. The format of \$inInvalidFields is that returned by the ValidateTextFields method.

This method validates that *inAddress* is valid using a regular expression that should cover the vast majority of possible email addresses

writeBold

version 3
modified version 4.0

```
writeBold(inText; inTest)
```

Parameter	Type	Description
inText	Text	→ Text to modify
inTest	Boolean	→ True to make inText strong

Discussion

This method writes *inText* to the response buffer. If *inTest* is *True*, *inText* is enclosed in a `` tag pair. `` is used because it is preferred by web designers. `` is a font weight, whereas `` conveys the meaning you want to impart to *inText* by changing it's style.

Batch

One of the most common tasks in a dynamic website is the display of columnar data following a query, much as you would display a listing form within 4D.

It is bad user interface design to display much more than one or two screens of data at one time. This means that you have to split the total number of data rows into batches and give the user a means of paging through them, much as you do when paging through the results of a google search.

The Batch library is designed to make the process of generating batches as simple as possible. In addition, it provides one-line methods for generating the HTML links necessary to go from one page to another.

Batches don't know anything about the nature of the rows they are batching. They have no access to the data. Their sole purpose in life is to deal with indexes into the rows of data. You must use those indexes to fetch the data from whatever data source you are using, be it a RowSet, table selection, or array.

Batch Attributes

A batch is an "object"-like thing that is represented by a collection. The collection contains several numeric attributes that control the behavior of the batch.

Note: You should consider batch attributes read-only. Don't try changing them once the batch is created unless you want unexpected results.

In order to use batches effectively, you need to understand their attributes:

Attribute	Description
size	Number of rows per batch
row_count	Number of rows managed by batch
start	Starting index of the batch
end	Ending index (inclusive) of the batch
orphan	Minimum batch size (maximum orphans)
overlap	How many rows to overlap in each batch

Creating a Batch

Batches are created from a set of rows. Those rows can come from a RowSet, a table selection, or an array. So to create a batch "object" you call the appropriate method in the Batch library.

To create a batch from a RowSet, you use the `newFromRowSet` method (surprise, surprise), like this:

```
$batch := Batch.newFromRowSet($rowset; $attributes)
```

To create a batch from a table selection, use the `newFromSelection` method:

```
$batch := Batch.newFromSelection(->[contacts]; $attributes)
```

To create a batch from an array, use the `newFromArray` method:

```
$batch := Batch.newFromArray($contacts; $attributes)
```

After calling these methods, you get back a batch “object” that contains the attributes listed above. Typically the only attributes you are interested in are the `start`, `end` and `row_count` attributes.

How Batches Are Calculated

To calculate the starting and ending index of a batch, five parameters are taken into account:

Parameter	Description
size	Number of rows in a batch
start	Starting index for batch
end	Ending index for batch
orphan	Minimum batch size
overlap	How many rows to overlap

To calculate the starting and ending indexes for the batch, the Batch library uses the following logic:

- If `start` is known and is > 0 , it is used as the starting index. Otherwise the starting index is 1.
- If the next batch would contain less than `orphan` elements, it is combined with the current batch.
- `overlap` is the number of rows shared by adjacent batches.
- If `size` is ≤ 0 , it is computed from `start` and `end`. Failing that, it is the default size.
- If `end` is not specified or is ≤ 0 , it is computed from the other parameters.

Batch Defaults

Typically, all of the batches in a given site will share the same size, minimum size (orphan), and overlap. The batch library allows you to set defaults for these values so that you do not have to specify them each time you create a batch.

When the Batch library is first imported, the following defaults are set:

Parameter	Default
size	10
orphan	2
overlap	0
start param	"bst"

You may set the defaults by calling `Batch.setDefaults`. Typically you would do this in the `On Application Start` event handler in `Active4D.a4l`.

If you do not specify the start index when creating a batch, you must pass one of the `newFromX` methods a collection handle or iterator which contains the default start param. Since the batch start index is usually kept in the query string, in a Fusebox application you would pass `$attributes`, as in the examples above, and in a non-Fusebox application you would usually pass `_query`.

Let's look at some examples to see how this works. Let's assume we have a `RowSet` that has 30 rows. We have a batch size of 10, a minimum batch size (orphan) of 3, and an overlap of 1. We will end up with 3 batches:

Batch #	Start - end indexes
1	1 - 10
2	10 - 19
3	19 - 30

Okay, let's analyze what's happening here. In the first batch, the starting index is 1, and the size is 10, so obviously we will get rows 1-10.

The last `<batch overlap>` rows of the each batch are repeated at the beginning of the next batch. So in the second batch, because we have an overlap of 1, the first row is row 10. A total of 10 rows takes us up to 19 as the ending index (not 20, that would be 11 rows).

Applying the same overlap trick to the third batch, we start at index 19 and add 9 to get a total of 10 rows. This takes us to an ending index of 28. But since the orphan (minimum) size is 3 and there are only 2 rows left, the final 2 rows are added to the end of the third batch, which gives us an ending index of 30. In this case examining the `size` attribute of the batch would reveal a size of 12.

Generating Batch Links

The main purpose of the Batch library is to help you generate links to the batches. Once you have created a batch, there are two ways you can do this: the hard way and the easy way.

The hard way is to iterate through the batches and build the links yourself, like this:

```
array longint($starts; 0)
$batch->getStarts($starts)
c_longint($i)

for ($i; 1; size of array($starts))
  if ($i = $starts) // current batch
    // build link for current batch
  else
    // build link for other batches
  end if
end for
```

That may not look so hard, but the code to build the links is not trivial. It's so non-trivial I'm too lazy to bother including it here, because all that work is done for you if you take the easy road, like this:

```
<%
c_text($prev; $batches; $next)
$needBatch := $batch->makeFuseboxLinks($prev; $batches; $next; \\
                                         $attributes)
%>

// later, when you need to display the links
<% if ($needBatch) %>
  <div id="batches">
    Go to page:&nbsp;&lt;%= $prev + $batches + $next %>
  </div>
<% end if %>
```

The “easy way” has a little more code here, but only because I am leaving so much out of the “hard way,” which in reality would have 10 times more code — code that you would end up copying and pasting all over the place.

The *makeFuseboxLinks* method and its non-Fusebox cousin, *makeLinks*, allow you to customize everything necessary to format the page links exactly the way you want, so you should never have to do it yourself.

Batch Defaults for Generating Links

To format the batch links, the Batch library needs the following parameters:

Parameter	Description	Default
previous	HTML CSS to use for “first” and “previous” link	« Previous batch-prev
separator	HTML to use between page numbers	
next	HTML CSS to use for “next” and “last” link	Next » batch-next
tag	HTML tag to use for the current batch	strong
max	Maximum batches to display	7

The defaults are set for you when the Batch library is imported. If you wish to change the defaults, you may do so with the *Batch.setDefaults* method.

When you call *makeFuseboxLinks* or *makeLinks*, to use the default values either omit parameters or pass empty strings or negative numbers, as is done in the example above.

Iterating Through Rows

Once you create a batch object, the start and end index for the current batch are accessible as items of the batch. To iterate through the current batch you would do something like this, given a RowSet called `$qryContacts`:

```
<%
// Make the batch using defaults
$batch := Batch.newFromSelection(->[contacts]; $attributes)

// Now make the RowSet from the subset of rows in the batch
$rs := RowSet.newFromSelection(->[contacts]; $map; \\  
    '$batch{"start"}:$batch{"end"}')

// Setup the link info, make the links
c_text($prev; $batches; $next)
$needBatch := $batch->makeFuseboxLinks($prev; $batches; $next; \\  
    $attributes)

if ($qryContacts->rowCount > 0)
%>
<p>Displaying
<%= '$batch{"start"}-$batch{"end"} of $batch{"row_count"} '%>
</p>
<table>
  <tr>
    <th>Name</th>
    <th>Phone</th>
  </tr>
  <%
    $row := $qryContacts->getRow
    $alt := true

    while ($qryContacts->next)
      $alt := not($alt)
  %>
  <tr<%=choose($alt; " class=\"row-alt\""; "")%>>
    <td><%= $row{"name"} %></td>
    <td><%= $row{"phone"} %></td>
  </tr>
<% end while %>
</table>
<% if ($needBatch) %>
  <div id="batches">
    Go to page:&nbsp;<%= $prev + $batches + $next %>
  </div>
<% end if %>
<% else %>
<p>There are no contacts that matched your query.</p>
<% end if %>
```

dumpDefaults

version 4.0

dumpDefaults

Discussion

This method produces a formatted dump of the current default parameters. For a discussion of the purpose of each parameter, see “Batch Defaults” on page 527 and “Batch Defaults for Generating Links” on page 529.

getDefaults

version 4.0

```
getDefaults(outSize; outMinSize; outOverlap; outMaxBatches; outPrevious;
outSeparator; outNext; outCurBatchTag; outStartParam)
```

Parameter	Type	Description
outSize	Number	← Receives batch size
outMinSize	Number	← Receives batch minimum size
outOverlap	Number	← Receives batch overlap
outMaxBatches	Number	← Receives maximum batches to display
outPrevious	Text	← Receives “previous batch” HTML
outSeparator	Text	← Receives batch separator HTML
outNext	Text	← Receives “next batch” HTML
outCurBatchTag	Text	← Receives HTML tag for current batch
outStartParam	Text	← Receives start index parameter name

Discussion

This method retrieves the current defaults for all values used by the Batch library. For a discussion of the purpose of each parameter, see “Batch Defaults” on page 527 and “Batch Defaults for Generating Links” on page 529.

getStarts

version 4.0

```
getStarts(self; outStarts)
```

Parameter	Type	Description
self	Batch	→ The batch object
outStart	Array Longint	← Receives batch starting indexes

Discussion

This method gets the starting indexes for every batch that can be generated for the set of rows referred to by *self*. The current element of *outStarts* is set to the current batch number.

makeFuseboxLinks

version 4.0

```
makeFuseboxLinks(self; ioPrevious; ioBatches; ioNext; inAttributes {; inXFA
  {; inPassedAttrs {; inCurBatchTag {; inStartParam {; inMaxBatches}}}}})
→ Boolean
```

Parameter	Type	Description
self	Batch	→ The batch object
ioPrevious	Text	↔ On entry, the HTML text for the first/previous batch link
ioBatches	Text	↔ On entry, the HTML text to insert between batch numbers
ioNext	Text	↔ On entry, the HTML text for the next/last batch link
inAttributes	Collection	→ Fusebox \$attributes collection
inXFA	Text	→ The XFA to link to
inPassedAttrs	Text	→ Semicolon-delimited list of attributes to pass in the links
inCurBatchTag	Text	→ Tag to enclose current batch in
inStartParam	Text	→ Query parameter name for start index
inMaxBatches	Number	→ Maximum number of batches to display
Result	Boolean	← True if links are needed

Discussion

This method is the same as **makeLinks**, but it calculates the *inLinkRef* parameter of that method from *inAttributes*, *inXFA*, and *inPassedAttrs*.

If *inXFA* is not passed in or is empty, it is set to the *circuit.fuseaction* of the current page.

If *inPassedAttrs* is passed in and is not empty, a query string is built from the semicolon-delimited list of attributes names, whose values are taken from *inAttributes*.

For the rest of the parameters and examples, see the discussion of *makeLinks*.

makeLinks

version 4.0

```
makeLinks(self; ioPrevious; ioBatches; ioNext; inLinkRef
  {; inCurBatchTag {; inStartParam {; inMaxBatches}}}) → Boolean
```

Parameter	Type	Description
self	Batch	→ The batch object
ioPrevious	Text	↔ First/previous batch link
ioBatches	Text	↔ Batch numbers
ioNext	Text	↔ Next/last batch link
inLinkRef	Text	→ The base URL to link to
inCurBatchTag	Text	→ Tag to enclose current batch in
inStartParam	Text	→ Query parameter name for start index
inMaxBatches	Number	→ Maximum number of batches to display
Result	Boolean	← True if links are needed

Discussion

This method generates the batch links for all batches that can be generated for the set of rows referred to by *self*.

There are three parts to the links generated: a “first/previous batch” link, the batch number links, and a “next/last batch” link. Each is generated separately to give you maximum flexibility in formatting the output.

On entry, *ioPrevious* should contain the encoded HTML text you would like to use for the “first/previous batch” link. If *ioPrevious* is empty, the default HTML “previous/first batch” text is used. If there are no previous batches, an empty string is returned.

Otherwise, if the text you pass in *ioPrevious* contains “^”, the text before it is used as the HTML to display for a “first batch” link, and the text after it is used as the HTML to display for a “previous batch” link. If there is no “^”, the text is used for a “previous batch” link.

If the text for the “first batch” or “previous batch” link contains a “|” (vertical bar), the text before it is the HTML to display, and the text after it is used as the CSS class name for the link.

On entry, *ioBatches* should contain the encoded HTML text you would like to insert between batch numbers, for example “ ”. If *ioBatches* is empty, the default HTML separator is used. If there are batches to display, on return *ioBatches* will contain HTML with links for each batch that should be displayed. If the text contains a “|” (vertical bar), the text before it is the text to display, and the text after it is used as the CSS class name for the links.

The *inCurBatchTag* parameter determines what HTML tag and CSS class, if any, will be used for the current batch number, which does not have a link. If this parameter contains a “|” (vertical bar), the text before it is the tag to use (without the enclosing <>), and the text after it is the CSS class name for the tag. If *inCurBatchTag* is empty, the default tag is used.

On entry, *ioNext* should contain the encoded HTML text you would like to use for the “next/last batch” link. If *ioNext* is empty, the default HTML “next/last batch” text is used. If there are no more batches, an empty string is returned.

Otherwise, if the text you pass in *ioNext* contains “^”, the text before it is used as the HTML to display for a “next batch” link, and the text after it is used as the HTML to display for a “last batch” link. If there is no “^”, the text is used for a “next batch” link.

If the text for the “next batch” or “last batch” link contains a “|” (vertical bar), the text before it is the HTML to display, and the text after it is used as the CSS class name for the link.

Note: Be sure to HTML encode the text you pass in *ioPrevious*, *ioBatches*, and *ioNext*. For example, if you would like the “previous batch” link to say “<< Previous ”, you would pass the text “<< Previous ”.

The *inLinkRef* parameter should contain the URL that will go in the *href* of the batch links. If you want to pass query parameters to the destination page, be sure to include them in the query string of this URL. You do not have to (and should not) include the batch start index, that is handled by this method.

Note: *inLinkRef* should be URL encoded.

If you want to customize the start index parameter name, pass its name in the *inStartParam* parameter. If *inStartParam* is omitted or is empty, the default is used.

Note: The standard start index parameter name “bst” is contained in the library constant *Batch.kStartParam*.

By default this method will generate HTML for all possible batches. If you want to limit the number of batch links that display at one time, pass the maximum number in the *inMaxBatches* parameter. If *inMaxBatches* is omitted or is < 0, the default is used. Passing a value of zero means there is no limit on the number of batches generated.

If there are more batches than the maximum, batches will be removed from the beginning or end of the list and the current batch will be centered within the list.

The result of this method will be *True* if there is more than one batch, which means you need to display the HTML generated by the method.

Example

Although it seems complicated, using this method is actually quite easy. Let’s take a look at the following scenario:

- We have a RowSet with 320 rows, broken up into 32 batches of 10 rows each with no overlap and a minimum batch size of 2.
- We want a “first batch” link of “First”, a “previous batch” link of “«Previous” with a CSS class of “batch-prev”, a “next batch” link of “Next»” with a CSS class of “batch-next”, and a “last batch” link of “Next”.
- We want the current batch number to use a `` tag with a CSS class of “batch-current”.
- The target URL will be the current page with a query parameter “cid” for the company ID, which we get from the current query params.
- We will accept the default batch start query parameter of “bst”.
- We want to show the default maximum of 7 batches at a time.

Given all of this information, the code to make all this happen is straightforward:

```
<%
$batch := Batch.newFromSelection(->[contacts]; _query)

// Set up the HTML
c_text($prev; $batches; $next)
$prev := "First^&laquo;Previous|batch-prev"
$next := "Next&raquo;|batch-next^Last"
$href := requested url + '?cid=_query{"cid"}'
$needBatches := $batch->makeLinks($prev; $batches; $next; \\  
                                $href; "span|batch-current")
%>

<A table with the row data>

<% if ($needBatches) %>
  <p><%= $prev + $batches + $next %></p>
<% end if %>
```

Six lines of code to generate and display the batching is not bad! If you are using Fusebox, assuming your display fuse receives an XFA for the batch links called `$XFA_onBatch`, you would use the `makeFuseboxLinks` method like this:

```
$batch := Batch.newFromSelection(->[contacts]; $attributes)

// Set up the HTML
c_text($prev; $batches; $next)
$needBatches := $batch->makeFuseboxLinks($prev; $batches; \\  
                                        $next; $attributes; $XFA_onBatch; "cid"; \\  
                                        "span|batch-current")
```

new**version 4.0**

`new(inRowCount; inCollection) → Longint`
`new(inRowCount; inSize {; inStart {; inEnd {; inOrphan {; inOverlap}}}) → Longint`

Parameter	Type	Description
<code>inRowCount</code>	Number	→ Row count of data batch is based on
<code>inCollection</code>	Collection	→ Collection from which to get default start index param
<i>OR</i>		
<code>inRowCount</code>	Number	→ Row count of data batch is based on
<code>inSize</code>	Number	→ Batch size
<code>inStart</code>	Number	→ Starting index into the array
<code>inEnd</code>	Number	→ Ending index into the array
<code>inOrphan</code>	Number	→ Maximum orphans
<code>inOverlap</code>	Number	→ How many rows to overlap
<code>Result</code>	Longint	← Batch “object”

Discussion

This method creates a new batch from row data whose size is *inRowCount*. For a full discussion of the parameters, see *newFromRowSet*.

newFromArray**version 4.0**

newFromArray(inArray; inCollection) → Longint
 newFromArray(inArray; inSize {; inStart {; inEnd {; inOrphan {; inOverlap}}}}) → Longint

Parameter	Type	Description
inArray	Array	→ Data source the batch is based on
inCollection	Collection	→ Collection from which to get default start index param
OR		
inArray	Array	→ Data source the batch is based on
inSize	Number	→ Batch size
inStart	Number	→ Starting index into the array
inEnd	Number	→ Ending index into the array
inOrphan	Number	→ Maximum orphans
inOverlap	Number	→ How many rows to overlap
Result	Longint	← Batch “object”

Discussion

This method creates a new batch from an array of row data contained in *inArray*. For a full discussion of the parameters, see *newFromRowSet*.

newFromRowSet**version 4.0**

newFromRowSet(inRowSet; inCollection) → Longint
 newFromRowSet(inRowSet; inSize {; inStart {; inEnd {; inOrphan {; inOverlap}}}}) → Longint

Parameter	Type	Description
inRowSet	RowSet	→ Data source the batch is based on
inCollection	Collection	→ Collection from which to get default start index param
OR		
inRowSet	RowSet	→ Data source the batch is based on
inSize	Number	→ Batch size
inStart	Number	→ Starting index into the array
inEnd	Number	→ Ending index into the array
inOrphan	Number	→ Maximum orphans
inOverlap	Number	→ How many rows to overlap
Result	Longint	← Batch “object”

Discussion

This method creates a new batch from the RowSet *inRowSet*.

There are two forms of this method. The first form receives a collection handle or iterator in the second parameter. The default start index parameter (usually “bst”) is taken from that collection, and its value is used as the start index. If the parameter is not found, the start index defaults to 1. The rest of the parameters used in the first form are taken from the defaults.

In the second form of this method, the number of rows desired in each batch must be passed in the *inSize* parameter.

By default a batch is constructed with a start index of 1 and an end index of *inSize*. To use a different start index, you can either leave *inStart* zero and pass a non-zero *inSize* and *inEnd*, or leave *inSize* zero and pass a non-zero *inStart* and *inEnd*.

If you would like to set the maximum number of orphans (i.e. the minimum batch size) in the last batch, pass a non-zero value in *inOrphan*.

If you would like to overlap one or more rows between batches, pass a non-zero value in *inOverlap*.

Note: For more information on these parameters, see “How Batches Are Calculated” on page 527.

The result of this method is actually a collection handle that you use with the other methods in the Batch library.

newFromSelection

version 4.0

```
newFromSelection(inTable; inCollection) → Longint
newFromSelection(inTable; inSize {; inStart {; inEnd {; inOrphan {; inOverlap}}}})
→ Longint
```

Parameter	Type	Description
inTable	Table pointer	→ Data source the batch is based on
inCollection	Collection	→ Collection from which to get default start index param
OR		
inTable	Table pointer	→ Data source the batch is based on
inSize	Number	→ Batch size
inStart	Number	→ Starting index into the array
inEnd	Number	→ Ending index into the array
inOrphan	Number	→ Maximum orphans
inOverlap	Number	→ How many rows to overlap
Result	Longint	← Batch “object”

Discussion

This method creates a new batch from a selection of records in the table pointed to by *inTable*. For a full discussion of the parameters, see *newFromRowSet*.

next**version 4.0**

next(self; ioLength; ioStart; ioEnd) → Boolean

Parameter	Type	Description
self	Batch	→ The batch object
ioLength	Number	↔ Batch size
ioStart	Number	↔ Starting index
ioEnd	Number	↔ Ending index
Result	Longint	← True if there are more batches

Discussion

On entry, *ioLength*, *ioStart* and *ioEnd* should be set to the current values for *self*. On exit they will be set to the values for the next batch, if any.

If there is another batch available, *True* is returned, else *False*.

Example

If you are determined to iterate through batches yourself, you will probably need to use this method like this:

```
$start := $batch{"start"}
$end := $batch{"end"}
$length := $batch{"size"}

while ($batch->next($length; $start; $end))
  // do something
end while
```

previous**version 4.0**

previous(self; ioLength; ioStart; ioEnd) → Boolean

Parameter	Type	Description
self	Batch	→ The batch object
ioLength	Number	↔ Batch size
ioStart	Number	↔ Starting index
ioEnd	Number	↔ Ending index
Result	Longint	← True if there are more batches

Discussion

On entry, *ioLength*, *ioStart* and *ioEnd* should be set to the current values for *self*. On exit they will be set to the values for the previous batch, if any.

If there is another batch available, *True* is returned, else *False*.

setDefault

version 4.0

setDefault(inSize; inMinSize; inOverlap; inMaxBatches; inPrevious;
inSeparator; inNext; inCurBatchTag; inStartParam)

Parameter	Type	Description
inSize	Number	→ Receives batch size
inMinSize	Number	→ Receives batch minimum size
inOverlap	Number	→ Receives batch overlap
inMaxBatches	Number	→ Receives maximum batches to display
inPrevious	Text	→ Receives “previous batch” HTML
inSeparator	Text	→ Receives batch separator HTML
inNext	Text	→ Receives “next batch” HTML
inCurBatchTag	Text	→ Receives HTML tag for current batch
inStartParam	Text	→ Receives start index parameter name

Discussion

This method retrieves the current defaults for all values used by the Batch library. For a discussion of the purpose of each parameter, see “Batch Defaults” on page 527 and “Batch Defaults for Generating Links” on page 529.

Breadcrumbs

Very often the data in a dynamic website is structured in a hierarchical fashion, and the user interface follows accordingly. For example, consider the following scenario:

- 1 The user selects a company from a list, which displays a list of departments.
- 2 The user selects a department, which displays the employees in that department.
- 3 The user selects an employee, which displays a list of projects the employee is working on.
- 4 The user selects a project, which displays a list of the work performed on the project by the employee.
- 5 The user selects a work item, which displays details about the work performed.

At the end of this process the user is five levels deep in the hierarchy. It is essential to provide some context for the user so he or she has some idea how they got to where they are and how to get back to where they came from.

The standard technique for doing this is called “breadcrumbs”. The idea is that as users journey through the site — especially if they are descending through a hierarchy — you leave a trail of “breadcrumbs” so that they can find their way back.

For example, if you go to www.aparajitaworld.com/store, then roll over the “Category” menu, and select the category “Plugins>Active4D>Deployment Licenses”, below the menu strip you will see the following:

```
Home > Plugins > Active4D > Deployment Licenses
```

These are the breadcrumbs. Going from left to right, we see the path you took to get to the page you are on. The text between each “>” is a breadcrumb along the way, and is also a link to jump directly to that point in the path.

The Breadcrumbs library gives you a simple interface for building and displaying breadcrumbs. By using this library you will save a lot of code.

Using Breadcrumbs

Using breadcrumbs is basically a three-step process:

- 1 Create a breadcrumbs object
- 2 Add breadcrumbs to the object
- 3 Write the breadcrumbs on your page

In practice step 2 can be a little complex, because you have to figure out exactly what URL is necessary to retrace your steps.

For example, let’s consider the following case using Fusebox:

- 1 From the main page (`fuseaction=app.main`), the user selects “View companies” (`fuseaction=companies.list`).

- 2 From the company list, the user selects a company to show the employees for that company (`fuseaction=employee.list`). The company id is passed in the query param `"cid"`.
- 3 From the employee list, an employee is selected (`fuseaction=employees.edit`). The employee id is passed in the query param `"eid"`.

Our Fusebox circuit structure is like this:

```
app
  companies
    employees
```

In a case like this, we can take advantage of the hierarchy of circuits and the fact that Fusebox will call `fbx_settings.a4d` in each circuit as it descends to the target circuit (`employees`). So `app/fbx_settings.a4d` will contain this code:

```
$breadcrumbs := Breadcrumbs.fuseboxNew("Home"; "app.main")
```

Now we have a new `breadcrumbs` object with a home link to the application's main page. Next we descend into the `companies` circuit. Since a company can be edited (`fuseaction=companies.edit`), we have to check the `fuseaction` in `app/companies/fbx_settings.a4d`:

```
$breadcrumbs->add("Companies"; "companies.list")

if ($fusebox{"isTargetCircuit"})
  if ($fusebox{"fuseaction"} = "edit")
    $breadcrumbs->add("Edit"; "companies.edit"; \\  
                    'cid=$attributes{"cid}")
  end if
end if
```

Finally we descend into the `employees` circuit. At this point the breadcrumbs are:

```
Home > Companies
```

In `app/companies/employees/fbx_settings.a4d` we do a check like in the `companies` circuit, because employees can be listed or edited:

```
$breadcrumbs->add("Employees"; "employees.list";  
'cid=$attributes{"cid}")

if ($fusebox{"isTargetCircuit"})
  if ($fusebox{"fuseaction"} = "edit")
    $query := a4d.web.collectionItemsToQuery($attributes; \\  
                                           "cid;eid")
    $breadcrumbs->add("Edit"; "employees.edit"; $query)
  end if
end if
```

When we arrive at the display fuse for the fuseaction *employees.edit*, the breadcrumbs have been completely built. Now in the *app/companies/employees/dsp_editForm.a4d* display fuse, we write out the breadcrumbs like so:

```
<% $breadcrumbs->write %>
```

The breadcrumbs library takes care of building the whole list, inserting a separator between each breadcrumb, and leaving out a link from the last one (since you are already there).

Customizing Breadcrumbs Appearance

Naturally you will want to have the breadcrumbs appear in a way that is consistent with the rest of your page. By default, the entire breadcrumbs display is wrapped in a div with the id "breadcrumbs", and the separator between breadcrumbs is the image */images/breadcrumb-separator.gif*.

If you choose not to use the default settings, you can change them with the methods *setSeparator* and *setDivId*. These settings apply globally to all breadcrumbs created by the Breadcrumbs library.

To control the appearance of the breadcrumb text and links, you must use CSS to define styles specific to the "breadcrumbs" div (or whatever id you choose for the div).

For example here is some CSS code that will format the breadcrumb elements:

```
#breadcrumbs
{
  padding: 3px;
  margin: 0;
  border: none;
  color: #333;
  font-family: Verdana, Helvetica, Arial, san serif;
  font-size: 1em;
  font-weight: normal;
  font-style: normal;
}

#breadcrumbs img
{
  margin: 0 5px;
  padding: 0;
  border: none;
}

#breadcrumbs a:hover
{
  background-color: #ccc;
  text-decoration: none;
  border-bottom: 1px solid #999;
}
```

add**version 4.0**

```
add(self; inText {; inURL {; inQuery{}}
```

Parameter	Type	Description
self	Breadcrumbs	→ The breadcrumbs object
inText	Text	→ Text to display
inURL	Text	→ URL/fuseaction for this breadcrumb
inQuery	Text	→ Query string for this breadcrumb

Discussion

This method adds a new breadcrumb to the end of the breadcrumbs list. If *self* was created with *fuseboxNew*, *inURL* should be a fully qualified fuseaction (circuit.fuseaction). Otherwise it should be the URL to target page for the breadcrumb. If you know the breadcrumb is the last one in the chain, you may omit *inURL* and *inQuery*.

inText is the text that will display when the breadcrumbs are written. It should be URL-encoded.

inQuery is the query string (if any) to be appended to the target URL. It should be URL-encoded, and may be supplied with or without a leading "?".

dumpLib**version 4.0**

```
dump
```

Discussion

This method dumps the library variables used for the default separator and div id.

fuseboxNew**version 4.0**

```
fuseboxNew(inHomeText {; inHomeAction {; inHomeQuery}}) → Longint
```

Parameter	Type	Description
inHomeText	Text	→ Text to display for the home breadcrumb
inHomeAction	Text	→ Fuseaction for the home breadcrumb
inHomeQuery	Text	→ Query string for the home breadcrumb
Result	Longint	← Breadcrumbs object

Discussion

This method creates a new breadcrumbs “object” for use with a Fusebox site and adds a breadcrumb with the values passed in.

For more information on the parameters, see “add” on page 545.

new**version 4.0**

```
new(inHomeText {; inHomeURL {; inHomeQuery}}) → Longint
```

Parameter	Type	Description
inHomeText	Text	→ Text to display for the home breadcrumb
inHomeURL	Text	→ URL for the home breadcrumb
inHomeQuery	Text	→ Query string for the home breadcrumb
Result	Longint	← Breadcrumbs object

Discussion

This method creates a new breadcrumbs “object” for use with non-Fusebox site and adds a breadcrumb with the values passed in.

For more information on the parameters, see “add” on page 545.

setDivId

version 4.0

`setDivId(inId)`

Parameter	Type	Description
inId	Text	→ div id to use for breadcrumbs

Discussion

This method sets the id of the div that will wrap breadcrumbs written with all future calls to the *Breadcrumbs.write* method.

For more information on how to use the div id, see “Customizing Breadcrumbs Appearance” on page 544.

setSeparator

version 4.0

`setSeparator(inSeparator)`

Parameter	Type	Description
inSeparator	Text	→ HTML to insert between breadcrumbs

Discussion

This method sets the HTML that will be inserted between breadcrumbs written with all future calls to the *Breadcrumbs.write* method.

For more information on how to control the appearance of the separator, see “Customizing Breadcrumbs Appearance” on page 544.

write

version 4.0

`write(self)`

Parameter	Type	Description
self	Breadcrumbs	→ The breadcrumbs object

Discussion

This method writes the breadcrumbs list to the response buffer wrapped in a div. All but the last breadcrumb are links.

For more information on how to control the appearance of the breadcrumbs, see “Customizing Breadcrumbs Appearance” on page 544.

fusebox

As software developers, we fight a continual battle to create order out of chaos and to minimize the amount of code we write (at least I do!). Towards this end we make methods, components, and frameworks, so that we might reduce the complexity of our applications, thereby allowing us to focus more clearly on the important problems at hand.

Fusebox is both a framework and a methodology that fulfills this promise. To describe it I will quote from the fusebox.org web site.

An Overview of Fusebox

Application developers face a daunting task: they must translate the often fuzzily-defined requirements for a new application into the rigid language of computers. While the Fusebox Lifecycle Process (FLiP) offers help in managing the project management aspects of creating a new application, what help is there available to developers approaching the technical challenges of creating and maintaining applications?

Application frameworks answer this question, offering pre-built (and pre-tested) code — a collection of services that can provide the architectural underpinnings for a particular type of application. As web development matures, web-based application frameworks allow the developer to concentrate more on meeting the business needs of the application and less on the “plumbing” needed to make that application work.

Fusebox is, by far, the most popular and mature web framework available for ColdFusion and PHP developers [*and now for Active4D developers!*]. The architecture of a Fusebox application is divided into various sections (“circuits” in Fusebox parlance), each of which has a particular focus. For example, the responsibility for ensuring that only authorized users have access to all or part of the application might fall under a Security circuit.

The Fusebox application architect defines these circuits, as well as the individual actions (“fuseactions”) that may be requested of it. When a fuseaction request is made of the application, the Fusebox machinery (the “Fusebox”) routes the request to the appropriate circuit, where the fuseaction is processed. This idea of encapsulation of responsibilities makes it easy for different functional circuits to be “plugged” into an application, making it possible to reuse code.

Within the individual circuit responsible for carrying out the requested fuseaction, the Fusebox architect specifies the individual files (“fuses”) needed to fulfill the fuseaction request. Thus, the Fusebox acts like a good manager, delegating tasks to appropriate departments where it is decomposed into individual tasks, each of which can be assigned to individuals to carry out.

Why Should I Use Fusebox?

There are a couple of outstanding reasons why you should start using Fusebox:

- It doesn't just promise to make web development easier, it actually fulfills that promise. With Fusebox you will write less code with less bugs that is easier to maintain.

- The guys who designed Fusebox are incredibly smart and have a lot of experience writing web applications. The chances are what they have come up with is a lot better than any sort of framework we could design.
- Last but certainly not least, Fusebox is the programming model of choice for Active4D. All demos and sample code are in Fusebox.

How Do I Learn Fusebox?

There are a number of good resources for Fusebox, including several books. You should definitely get the Fusebox 3 books and start there. Although the code in those books is in ColdFusion, you should have no problem translating it to Active4D.

A complete list of Fusebox resources can be found at:

<http://www.fusebox.org/index.cfm?fuseaction=fusebox.resources>

Active4D provides many Fusebox code examples that illustrate key techniques. Be prepared to spend a week trying to understand how Fusebox works. Don't give up — at some point it all becomes clear, and you realize that it is actually incredibly simple.

Active4D's Fusebox Implementation

Currently Active4D provides a very robust implementation of Fusebox 3. The reference implementation of Fusebox is now at version 5.5. Perhaps Active4D's Fusebox implementation will be updated in the future, but doing so would actually break existing Fusebox code, so for now Fusebox 3 is more than sufficient.

Differences Between the ColdFusion and Active4D Implementation

As you are reading the Fusebox books, which are based on ColdFusion, you will notice some differences between the ColdFusion implementation of Fusebox and the Active4D implementation. Here is a quick guide to translating between the two versions:

- **Syntax differences:** Where ColdFusion talks about structures, Active4D will use collections. When you see a ColdFusion structure reference like *Fusebox.circuit*, the Active4D equivalent is *\$fusebox{"circuit"}*.
- **Fusebox variable differences:** ColdFusion sets a boolean variable *isCustomTag* in the *Fusebox* structure to indicate if a nested fuseaction is being called. In Active4D, you examine the collection item *\$fusebox{"isNestedCall"}*.

Active4D also defines several new variables in *\$fusebox*:

Variable	Description
appPath	Contains the full Unix path to the root circuit of the application. This is needed when calling the core directly.
fqfa	Contains the fully qualified fuseaction, i.e. <i>circuit.fuseaction</i>
isNestedCall	Boolean variable set to true when the core is called by sendFuseaction

Variable	Description
nestLayouts	When set to false by a circuit, prevents its layouts from being nested in parent circuit layouts
self	contains the name of the single point of entry into fusebox, usually "index.a4d".

- **Fusedoc differences:** The `<structure></structure>` element should be called `<collection></collection>` in Active4D fusedocs.
- **Convenience methods:** Active4D adds a few convenience methods to the fusebox library (discussed below) for creating URLs suitable for use with fusebox.

Circuit libraries and initializers

In Active4D circuits are truly self-contained and self-initializing, which increases your ability to design generic circuits.

Each circuit can have its own libraries and initialization script. At startup, after Active4D.ini is read, the web root directory for each virtual host is recursively traversed. Each directory that contains the file `fbx_switch.a4d` is assumed to be a Fusebox circuit. If it is a circuit, the following two steps are taken:

- 1 Any files whose filename extension matches the configured library extension exactly (case is significant) are imported as libraries. Since the `__load__` method will be executed when the library is imported, you can perform initialization of circuit libraries at startup.
- 2 If the file `__init__.a4d` is found, it is executed. Note that the code in this script must be embedded in `<% %>` tags, and that any output is ignored. This script gives you a chance to perform circuit-level initialization. Circuit libraries may be referenced within this script.

Note: It is your responsibility to ensure that no two circuit libraries in all circuits share the same name with other circuit libraries or with global libraries.

Configuring Fusebox

By default the name of the fuseaction query parameter is "fuseaction". In actual fact this name is not hard-coded into the `fusebox` library, but is defined as the library constant `fuseaction` in the `fusebox.conf` library.

If you would like to change the fuseaction query parameter name, you can do so by changing the define in the `fusebox.conf`. If you want to refer to the fully qualified `circuit.fuseaction` within your fuses, you should always do so in this way:

```
$fuseaction := $fusebox{fusebox.conf.fuseaction}
```

core**version 3**
modified version 4.0

core(inRootPath {; ioAttributes {; inNestedCall}})

Parameter	Type	Description
inRootPath	Text	→ Path to the directory of index.a4d
ioAttributes	Collection	↔ The request context
inNestedCall	Boolean	→ True if called from sendFuseaction

Discussion

This method is the Active4D Fusebox core file. You should never have to call this method directly.

getURLFactory**version 4.5**

getURLFactory → Text

Parameter	Type	Description
Result	Text	← Current URL factory method

Discussion

This method returns the name of the method which will be called to create URLs when *fusebox.makeURL* is called.

For more information on URL factory methods, see “setURLFactory” on page 555.

handleError**version 4.0**
deprecated v6.0

handleError(inTarget {; inQuery})

Parameter	Type	Description
inTarget	Text	→ The fully qualified circuit.fuseaction to call, or a full URL+query to redirect to
inQuery	Text	→ Query string

Discussion

This method has been deprecated in favor of **handleErrorInline**, it should no longer be used.

This method is meant to be called from the error page configured in Active4D.ini, or the error page set by a call to **set error page**.

If *inTarget* begins with `"/`, it is considered to be a full URL + query string (it should be URL-encoded) to redirect to. Otherwise it is considered to be a fully qualified circuit.fuseaction, which will be passed along with *inQuery* to *fusebox.makeURL* to construct the URL to redirect to. If *inQuery* is passed, it should already be URL-encoded.

Before redirecting to the target URL, the standard `$a4d_err_x` error variables are put in the session as `session{"a4d_err_x"}`. In addition, the full requested URL and referer are put in the session as `"a4d_err_url"` and `"a4d_err_referer"` respectively.

To ensure proper cleanup, the target fuseaction should call *fusebox.postHandleError* after using the session items created by this method.

For a discussion of how to handle errors in Fusebox, see "Custom Error Handling in Fusebox" on page 612.

handleErrorInline

v6.0

```
handleErrorInline(self; inTarget {; inAttributes})
```

Parameter	Type	Description
self	Collection	→ The <i>\$fusebox</i> collection
inTarget	Text	→ The full circuit.fuseaction to call to handle errors
inAttributes	Collection	→ Attributes to send to the target fuseaction

Discussion

This method is meant to be called from the error page configured in Active4D.ini, or the error page set by a call to **set error page**.

inTarget should be a fully qualified circuit.fuseaction, which will be sent using *fusebox.sendFuseaction* along with *inAttributes*. If *inAttributes* is not passed, a new collection is created and passed to the fuseaction.

The standard `$a4d_err_x` error variables are put in the *\$attributes* passed to the target fuseaction as `$attributes{"a4d_err_x"}`. In addition, the full requested URL and referer are put in *\$attributes* as `"a4d_err_url"` and `"a4d_err_referer"` respectively.

Unlike the deprecated **handleError** method, this method does not redirect and thus maintains the response status and execution context in which the error occurred. It also does not require a session to be created.

]For a discussion of how to handle errors in Fusebox, see "Custom Error Handling in Fusebox" on page 612.

invalidAction

v6.0

invalidAction(self)

Parameter	Type	Description
self	Collection	→ The <i>\$fusebox</i> collection

Discussion

This is a convenience method you can call when an `fbx_switch.a4d` catches an unhandled action. It throws a user error with the error code `kInvalidActionError` and an informative message.

Example

```
case of
  :($action = "main")
    include("dsp_main.a4d")

  else
    $fusebox->invalidAction
end case
```

isFuseboxRequest

version 4.0

isFuseboxRequest → Boolean

Parameter	Type	Description
Result	Boolean	← Query string

Discussion

This method returns *True* if the current request is being handled by Fusebox, *False* if not.

makeURL

version 4.0
modified version 4.5

makeURL(inFuseaction {; inQuery}) → Text

Parameter	Type	Description
inFuseaction	Text	→ The fully qualified circuit.fuseaction to call
inQuery	Text	→ Query string
Result	Text	← URL suitable for use with Fusebox

Discussion

This method creates a Fusebox URL suitable for use with links, form actions, redirects, etc.

If *inQuery* is passed, it should already be URL-encoded.

If a URL factory method has been set with *fusebox.setURLFactory*, that method will be called with the arguments *inFuseaction* and *inQuery*.

postHandleError

version 4.0
deprecated v6.0

postHandleError

Discussion

If you are using the new **handleErrorInline** method, this method is no longer necessary and is deprecated.

This method should be called from the target fuseaction of the *fusebox.handleError* method, after the session items created by *fusebox.handleError* are used.

sendFuseaction

version 3
modified version 4.0

```
sendFuseaction(self; inFuseaction {; ioAttributes})
```

Parameter	Type	Description
self	Collection	→ The <i>\$fusebox</i> collection
inFuseaction	Text	→ The fully qualified <i>circuit.fuseaction</i> to call
ioAttributes	Collection	↔ The request context

Discussion

When you want to call a fuseaction within the fuse of another fuseaction, use this method. Typically you would call it like this:

```
// Insert a calendar, pass today's date in the attributes
$attrs := new collection("date"; \\  
    string(current date; MM DD YYYY Forced))
$fusebox->sendFuseaction("calendar.view"; $attrs)
```

ioAttributes should contain the same items that can validly be passed to the target fuseaction either through a query string or through form variables.

Because the new fuseaction is running within a method call, you will not have access to any of the local variables used within the calling fuseaction. To pass values to the nested fuseaction, use *ioAttributes* or the built in **_request** collection.

setURLFactory

version 4.5

```
setURLFactory(inMethod) → Text
```

Parameter	Type	Description
inMethod	Text	→ The method to delegate URL creation to
Result	Text	← Previous URL factory method

Discussion

If you want to use your own URL style, use this method to set the name of a method which will be called to create URLs when *fusebox.makeURL* is called. Your factory method should take the same parameters as *fusebox.makeURL* and return the full URL.

If *inMethod* is empty, the default URL form is used.

The previous URL factory method is returned.

Example

Let's assume that instead of the default URL form:

```
/index.a4d?action=<action>.<action>;<query>
```

we want to use the form:

```
/<action>/<action>?<query>
```

We have a *makeUrl* method in a library called *mylib* which can create URLs in the new form. The method looks like this:

```
method "makeUrl"($inFuseaction; $inQuery = "")

  $circuit := slice string($inFuseaction; "."; $action)
  $url := "%s/%s" % ($circuit; $action)

  if (length($inQuery) > 0)
    $url += "?" + $inQuery
  end if

  return ($url)

end method
```

We can easily use our new URL style without changing all of our calls to *fusebox.makeURL* by setting a custom URL factory method. In this case we would call the following code in the *On Application Start* method of the Active4D library:

```
fusebox.setURLFactory("mylib.makeUrl")
```

fusebox.head

A Fusebox application is made up of many different circuits that work together. The Fusebox methodology (and good programming practice) demands that these circuits be as independent as possible. Ideally they should know nothing about each other.

In addition, Fusebox also tries to separate the presentation of a site from the logic by using layouts. These layouts typically are responsible for writing the HTML header in a page. And like the circuits, the layouts should know little or nothing about the rest of the application. But what happens if the circuits need to write things into the HTML header, such as meta tags, stylesheet includes, Javascript includes, etc. How can we allow circuits to write to the header without the header knowing about them?

This library facilitates the decoupling of circuits and layouts by providing a means for registering information that should go in the HTML header. The layouts can then use a method in this library to write that info without knowing anything about the circuits themselves.

How To Use This Library

There two places you should be using this library: in *fbx_settings.a4d*, and in the actual layout files themselves. In *fbx_settings.a4d* you add calls to set the window title, add stylesheet and Javascript references, and declare meta tags. In your layout file (or a file included by the layout file), you call **fusebox.head.write** somewhere within the <head> section of the document.

For example, let's assume a calendar circuit needs to load a stylesheet and a Javascript file. In the calendar circuit's *fbx_settings.a4d*, we add the following code:

```
fusebox.head.addCSS($fusebox; "css/calendar.css")
fusebox.head.addJS($fusebox; "js/calendar.js")
```

In the layout file *lay_main.a4d*, we put the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type"
      content="text/html; charset=utf-8">
<% fusebox.head.write %>
</head>
```

Because you are adding header elements in *fbx_settings.a4d*, they are added from the most generic to the most specific as Fusebox traverses from the root circuit to the target circuit.

Note: This library creates items in the *_request* collection whose keys begin with "fusebox.head.". Be sure not to use such keys yourself for any *_request* items.

addCSS

version 4.0

addCSS(inFusebox; inURLs)

Parameter	Type	Description
inFusebox	Collection	→ <i>\$fusebox</i>
inURLs	Text	→ Semicolon-delimited list of URLs to stylesheets

Discussion

This method adds one or more URLs that will go into the *href* attribute of a `<link>` tag to load a CSS stylesheet. To pass more than one URL, concatenate them together with semicolons. The URLs will be URL-encoded for you.

If the URL of a stylesheet is a relative path, it is assumed it is relative to the directory of the current circuit. This makes it much easier for you to specify circuit-relative paths. If you want to use a path which is relative to the Fusebox root, you should do something like this:

```
$path := join paths($fusebox{"rootPath"}; "path/to/styles.css")
```

If you want the stylesheets to apply to a given media, prefix the filename of the stylesheet with "@", followed by the media type.

If you want a stylesheet to be an alternate stylesheet, add the alternate stylesheet title after the filename, separated by a colon.

If you want to pass verbatim text, pass "!" followed by the text.

If you want to add attributes to the *link* tag which is generated, append "|" followed by the attributes (and values if necessary) to the filename.

Examples

Let us assume a calendar circuit needs to have access to a stylesheet *calendar.css* that is within a *css* directory within the circuit directory. To add this stylesheet to the header, we would execute the following code in the *fbx_settings.a4d* of the calendar circuit:

```
fusebox.head.addCSS($fusebox; "css/calendar.css")
```

If more than one path needs to be added, we can concatenate them together using the **concat** command:

```
fusebox.head.addCSS($fusebox; concat(";"; \\
  "css/calendar.css"; \\
  "css/@screen:calendar_small.css:small text"; \\
  "css/@screen:calendar_large.css:large text"; \\
  "css/@print:calendar_print.css"; \\
  "css/event.css"))
```

Note in the above example that *calendar_small.css* and *calendar_large.css* are screen media alternate stylesheets with the title “small text” and “large text” respectively. The stylesheet *calendar_print.css* will only be used when printing.

Assuming the path from the web root to the calendar circuit is *portlets/calendar*, a call to **fusebox.head.write** will result in the following comments and tags being written:

```
<!-- css added by calendar circuit ->
<link rel="stylesheet" type="text/css"
      href="/portlets/calendar/css/calendar.css" />
<link rel="alternate stylesheet" type="text/css"
      href="/portlets/calendar/css/calendar_small.css"
      media="screen" title="small text" />
<link rel="alternate stylesheet" type="text/css"
      href="/portlets/calendar/css/calendar_large.css"
      media="screen" title="large text" />
<link rel="stylesheet" type="text/css"
      href="/portlets/calendar/css/calendar_print.css"
      media="print"/>
<link rel="stylesheet" type="text/css"
      href="/portlets/calendar/css/event.css" />
<!-- end calendar circuit css -->
```

Now let’s suppose we have to create a specialized stylesheet for IE6 to get around it’s bugs. We can do that by using a conditional stylesheet include, like this:

```
fusebox.head.addCSS(concat(";"; \\
    "<!--[if lt IE 7]>"; \\
    "/css/ie6.css"; \\
    "!<![endif]-->"))
```

This would generate the following in the header:

```
<!--[if lt IE 7]>
<link rel="stylesheet" type="text/css" href="/css/ie6.css" />
<![endif]-->
```

addDumpStyles

version 4.0

addDumpStyles

Discussion

This method sets a flag which will cause the method **a4d.debug.writeDumpStyles** to be called when **fusebox.head.write** is called.

Normally you do not need to call this method, it is called by the methods in the *a4d.debug* library. If you plan to use the Active4D dump styles in a fusebox application, use this method, since the styles get written to the page header where they belong.

addJavascript

version 4.0

addJavascript(inFusebox; inURLs)

Parameter	Type	Description
inFusebox	Collection	→ <i>\$fusebox</i>
inURLs	Text	→ Semicolon-delimited list of URLs to Javascript files

Discussion

This method is a synonym for *addJS*.

addJS

version 4.0

addJS(inFusebox; inURLs)

Parameter	Type	Description
inFusebox	Collection	→ <i>\$fusebox</i>
inURLs	Text	→ Semicolon-delimited list of URLs to Javascript files

Discussion

This method adds one or more URLs that will go into the *src* attribute of a `<script>` tag to load a Javascript file. To pass more than one URL, concatenate them together with semicolons. The URLs will be URL-encoded for you.

If the URL of a Javascript file is a relative path, it is assumed it is relative to the directory of the current circuit. This makes it much easier for you to specify circuit-relative paths. If you want to use a path which is relative to the absolute web root, begin the path with `"/`. If the Fusebox root (where *index.a4d* is located) is within a subdirectory of the web root, you have to build the path to the Javascript file like this:

```
$path := join paths($fusebox{"rootPath"}; "path/to/script.js")
```

If you want to pass verbatim text, pass `!"` followed by the text.

If you want to add attributes to the *script* tag which is generated, append `"|"` followed by the attributes (and values if necessary) to the filename.

Example

Let us assume a calendar circuit needs to have access to a Javascript file *calendar.js* that is within a *js* directory within the circuit directory. To load this Javascript file in the header, we would execute the following code in the *fbx_settings.a4d* of the calendar circuit:

```
fusebox.head.addJS($fusebox; "js/calendar.js")
```

If more than one path needs to be added, we can concatenate them together like this:

```
fusebox.head.addJS($fusebox; \\
concat(";"; "js/calendar.js"; "js/events.js"; "/js/help.js"))
```

Assuming the path from the web root to the calendar circuit is *portlets/calendar*, a call to **fusebox.head.write** will result in the following tags being written:

```
<!-- Javascript added by calendar circuit -->
<script type="text/javascript"
src="/portlets/calendar/js/calendar.js">
</script>
<script type="text/javascript"
src="/portlets/calendar/js/event.js">
</script>
<script type="text/javascript" src="/js/event.js"></script>
<!-- end calendar circuit Javascript -->
```

Now let's suppose we have to include a Javascript file only on IE6 to enable it to display PNG images with transparency. In addition, we have to add the "defer" attribute to the *script* tag, because the file must not be executed when it is loaded.

We can accomplish this by using a conditional include, like this:

```
fusebox.head.addJS(concat(";"; \\
"!<!--[if lt IE 7]>"; \\
"/js/pngfix.js|defer"; \\
"!<![endif]-->"))
```

This would generate the following in the header:

```
<!--[if lt IE 7]>
<script type="text/javascript" src="/js/pngfix.js" defer>
</script>
<![endif]-->
```

Note we added the "defer" attribute by appending "|defer" to the file path.

addMetaTag

version 4.0

`addMetaTag(inFusebox; inTags)`

Parameter	Type	Description
<code>inFusebox</code>	Collection	→ <i>\$fusebox</i>
<code>inTags</code>	Text	→ Vertical bar-delimited list of meta tags

Discussion

This method adds one or more `<meta>` tags to the header. Only the content of the meta tag should be passed, not the meta tag itself. To pass more than one meta tag, concatenate them together with vertical bars (`|`). The meta tag content will not be URL encoded for you, it must be URL encoded by you if necessary.

Example

When a new user comes to the site, we redirect to a checker circuit which determines which browser is being used, as well as if Javascript and cookies are enabled. In order to do the cookie and Javascript checking, the circuit needs to put a meta refresh tag in the header:

```
$url := "http://" + join paths(request info{"host"}; \\  
                                fusebox.makeURL("checker.checkJS")  
$tag := 'http-equiv="refresh" content="1;URL=$url" '  
fusebox.head.addMetaTag($fusebox; $tag)
```

If our current host is `"my.host.com"` and `fusebox.conf.fuseaction` is `"action"`, a call to **`fusebox.head.write`** will result in the following tag being written:

```
<!-- meta tags added by checker circuit -->  
<meta http-equiv="refresh"  
content="1;URL=http://my.host.com/index.a4d?action=checker.JS" />  
<!-- end checker circuit meta tags -->
```

getTitle

version 4.0

`getTitle`

Parameter	Type	Description
<code>inTitle</code>	Text	→ Window title

Discussion

This method sets the title of the window, which will be written out as the `<title>` tag when **`fusebox.head.write`** is called. The text will be HTML encoded when it is written, you should not encode it yourself.

setTitle

version 4.0

```
setTitle(inTitle)
```

Parameter	Type	Description
inTitle	Text	→ Window title

Discussion

This method sets the title of the window, which will be written out as the <title> tag when **fusebox.head.write** is called. The text will be HTML encoded when it is written, you should not encode it yourself.

In a Fusebox application, you should call this method in the root circuit's *fbx_settings.a4d*, then you can specialize the title in other circuits by calling it in other circuits' *fbx_settings.a4d* or display fuses. The call closest to the target circuit will ultimately set the title.

write

version 4.0

```
write
```

Discussion

This method writes the tags specified by the **add<type>** methods and the **setTitle** method. When writing the tags, they are separated by circuit and their source is noted in HTML comments to aid in debugging your code.

RowSet

In the course of building a dynamic, database-driven web site, one of the most common tasks is to perform a query and then display or edit the results of that query. Such actions always have the following attributes:

- The result of the query is an ordered *set* of one or more *rows* in the database.
- Each row consists of one or more *columns*.
- To display the set of rows you must *iterate* over each row in order.
- To display a single row you must access the *column data* for that row.
- The *data source* of the column data typically consists of fields within a selection of records or elements within an array.

Enter the RowSet

A RowSet is a data structure that encapsulates all of the attributes listed above. The RowSet library is a set of methods that provides an API for working with a RowSet in an abstract and consistent manner.

There are several key advantages to using RowSets:

- Simple syntax for working with rows
- Uniform API no matter what the source of the row data is
- Ability to use dummy data if real table data is not available
- You can embed complex queries and data manipulations into the RowSet specification, much like a SQL query

At the heart of a RowSet is the data source. The data source acts as the conduit between the actual row data and the RowSet API. Currently there are three possible data sources for a RowSet:

- Selection of records
- Arrays
- Delimited text data

No matter what the data source is, once a RowSet is created the code that uses a RowSet will not change.

In addition to insulating you from the data source, RowSets also insulate you from changes to table, field, and array names. RowSets accomplish this in several ways:

- When you create a RowSet, you map virtual column names to the data source columns.
- You can access RowSet data by index, with the order being determined by the order in which you declare the columns in the mapping.

This architecture means that changes to the underlying data source will affect *only* the one or two lines of code where you create the RowSet and declare the column/data mapping.

For example, let's say that you want to display a list of people. The list will have the following columns:

- name
- phone
- email

Originally the data will come from these fields:

- [People]Name
- [People]Phone
- [People]Email

To create a RowSet from this data, you would do something like this:

```
$map := ""
name: [People]Name;
phone: [People]Phone;
email: [People]Email ""

$rs := RowSet.newFromSelection(->[People]; $map)
```

In this case the logical column names happen to correspond to the physical field names in the structure. But what happens if the structure changes (as it often does)? For example, let's say that the [People]Name field is renamed to [People]FullName, new phone number and email address fields are added, and the original fields are renamed, such that the data source for our "name", "phone" and "email" columns now are:

- [People]FullName
- [People]HomePhone
- [People]Email1

If you were using field names directly in your pages, you would have to do a search and replace in your entire site to make sure existing code doesn't break. On the other hand, if you are using a RowSet, only the single line of code that creates the column/field mapping will change, like this:

```
$map := ""
name: [People]FullName;
phone: [People]HomePhone;
email: [People]Email1 ""
```

All of the rest of the code that uses the RowSet will remain unaffected. "Okay," you say, "that's fine, but I don't see what the big deal is. I still have to change the field names in all of my mappings." While this is true, insulation from structure changes is not the only benefit you get from RowSets.

Let's suppose that at some point you decide not to access the selection directly, but for one reason or another you fetch (or build) the data into these arrays:

- \$names
- \$phones
- \$emails

Without RowSets you would be in for a lot of work, because the whole paradigm for accessing the data has changed. At best you will have a lot of global find and replace operations to perform, and the chances of it happening successfully are low.

On the other hand, if you use RowSets you only need to change the one line of code that creates the RowSet, like this:

```
$map := ""
name: $names;
phone: $phones;
email: $emails ""

$rs := RowSet.newFromArrays($map)
```

None of the code that works with the RowSet will have to change! But it gets better.

When you are developing your web site, you may not have access to the database, either because it has not yet been finalized or because you are working remotely. In that case you want to have some sample data so you can test your code, but where do you get it from?

With RowSets it is easy. You can create a RowSet from delimited text data, which could be read in from a file. To use this as your data source instead of selections or arrays, you need only change the RowSet creation code, like this:

```
$rs := RowSet.newFromData($myData)
```

Again, none of the code that uses the RowSet will have to change.

Using RowSets

Once you have created a RowSet, to iterate through its rows you simply do this:

```
while ($rs->next)
  // Show data
end while

// sometimes a for loop is preferable

for ($i; $rs->getStart; $rs->getEnd)
  $rs->gotoRow($i)
  // Show data
end for
```

How you choose to display the row data is up to you. You have three ways of accessing column within a row:

- By getting the row data collection with *getRow* and indexing that by column name
- By column index using *getData*
- By column name using *getData*

Of the three techniques, the first is recommended, because it results in clearer (and faster) code. Here is an example of the first technique:

```
<%
  $row := $rs->getRow

  while ($rs->next)
%>
<tr>
<td><% =$row{"name"} %></td>
<td><% =$row{"phone"} %></td>
<td><% =$row{"email"} %></td>
</tr>
<% end while %>
```

This is by far the simplest technique. However, there may be times when you are programmatically creating columns and it is easier to iterate through the columns by index, like this:

```
<%
  $numColumns := $rs->columnCount

  while ($rs->next)
%>
<tr>
<% for ($i; 1; $numColumns) %>
<td><% =$rs->getData($i) %></td>
<% end for %>
</tr>
<% end while %>
```

You may not like using indexes, since you can't see what column you are using without referring to the column mapping, so you can use column names instead. Here's an example of using column names to access column data:

```
<% while ($rs->next) %>
<tr>
<td><% =$rs->getData("name") %></td>
<td><% =$rs->getData("phone") %></td>
<td><% =$rs->getData("email") %></td>
</tr>
<% end while %>
```

That's about all there is to iterating through and accessing row data in a RowSet. It's about as easy as it can get. And remember...you can change your data source and none of your iterating code will have to change!

Subsetting Source Rows

In most cases you actually want a RowSet to represent a subset of the source data rows. For example, if you are displaying a selection of 100 records, you will typically batch them into groups of 10 or so rows. (And if you are not, you might want to consider doing so.)

If you know in advance you are going to show a subset of the source rows, you can indicate the subset when creating the RowSet. If the RowSet is cached or is created from arrays or delimited data, this can dramatically improve performance, because the RowSet will only store and process the given subset of rows.

There are two ways to indicate a row subset:

- Pass a Batch when creating the RowSet. The Batch start and end indexes will become the start and end rows of the RowSet.
- Pass the start and end indexes as a delimited pair in the form "<start>:<end>".

For more information on using row subsets, see the *newFrom<Source>* method descriptions below.

RowSet Cursors

Every RowSet has a *cursor* which keeps track of the current row. Unlike 4D query commands which automatically go to the first record in a selection, when a RowSet is first created the cursor is *before* the first row of the RowSet.

Like the current record in a selection, the cursor can be in one of three states:

- **before first:** This corresponds to **Before selection** in 4D. This is the state you ordinarily start in when iterating forward through a RowSet.
- **within:** The cursor index is within the range 1 to the number of rows in the RowSet if the RowSet has been created with the entire set of source data. If the RowSet represents a subset of the source data, the cursor index is within the range *<subrange start>* to *<subrange end>* inclusive.
- **after last:** This corresponds to **End selection** in 4D. This is the state you ordinarily start in when iterating backward through a RowSet.

There is a full suite of methods that allow you to move the cursor and determine its current state.

Persistent RowSets

There is a certain amount of overhead involved in creating a RowSet. For data sets that change infrequently or not at all, it would be nice if you could avoid the overhead of creating a new RowSet on every request.

By passing a name to one of the `newFrom<Source>` methods, you create a *persistent* RowSet: one that remains in memory until you specifically clear or overwrite it. Once a persistent RowSet is created, you can access it by name in subsequent requests.

In addition, because you can set a timeout for a RowSet, you can easily implement a refresh interval. For example, let's say you want to create a persistent RowSet for a list of product categories, since they change very infrequently, and you would like the RowSet to timeout every 8 hours. You would do something like this:

```
all records([categories])
order by([categories]; [categories]name)

$map := ""
id: [categories]id;
name: [categories]name""

$qryCategories := RowSet.newFromSelection(->[categories]; \\  
$map; -1; ""; "categories"; 60 * 8)
```

By passing a name as the next to last parameter when creating a RowSet, you make that RowSet persistent. The number after the name is the RowSet timeout in minutes.

To retrieve a persistent RowSet, you simply do this:

```
$qryCategories := RowSet.persistent("categories")
```

If the given persistent RowSet does not exist, zero is returned.

To see if a RowSet has timed out, you do this:

```
if ($qryCategories->timedOut)
```

Putting it all together, your code to create a persistent RowSet will follow this sort of pattern:

```
$qryCategories := RowSet.persistent("categories")

if ($qryCategories # 0)
  if (not($qryCategories->timedOut))
    return
  end if
end if

all records([categories])
order by([categories]; [categories]name)

$map := ""
id: [categories]id;
name: [categories]name""

$qryCategories := RowSet.newFromSelection(->[categories]; \\  
$map; -1; ""; "categories"; 60 * 8)
```

Which RowSet to Use

To get the most out of RowSets, you should know which kind of RowSet to create. Each RowSet creation method is designed to fulfill a specific need.

- **Data comes from the database:** If the source of your data is the database, use *newFromSelection* or *newFromCachedSelection*. Using row callbacks and column expressions, you can do very sophisticated queries and data manipulations with very little work. If you need to search in a RowSet or iterate through its rows more than once, consider using *newFromCachedSelection*.

If you need to use a cached selection and one or more columns must be calculated, consider using **SELECTION TO ARRAY** to load the non-calculated columns and then using *setColumnArray* to add the calculated columns.

If you need to directly access a blob field, you must use *newFromSelection*, since all of the other RowSets are array-based, and there are no blob arrays in 4D. To get around this you can use a column expression that extracts the data you need from the blob, in which case *newFromCachedSelection* can be used.

- **Data comes from arrays:** If you need to build arrays through complex queries and manipulations of the data, consider using *newFromCachedSelection* with a row callback and column expressions. Usually that can accomplish what you need but it does most of the work for you. If you just need to create a RowSet with one or more calculated columns, you can load the non-calculated columns using **SELECTION TO ARRAY** and then use *setColumnArray* to add the calculated columns. For more information on this strategy, see “*setColumnArray*” on page 590.

If the ultimate source of your data is arrays, obviously you must use *newFromArrays*.

afterLast

version 3

`afterLast(self)`

Parameter	Type	Description
self	RowSet	→ RowSet reference

Discussion

This method moves the cursor of *self* to just after the last row in the RowSet. After this call *isAfterLast* will return *True* and *previous* will move to the last row if there is one.

beforeFirst

version 3

`beforeFirst(self)`

Parameter	Type	Description
self	RowSet	→ RowSet reference

Discussion

This method moves the cursor of *self* to just before the first row in the RowSet. After this call *isBeforeFirst* will return *True* and *next* will move to the first row if there is one.

clearPersistent

version 4.0

`clearPersistent(inName)`

Parameter	Type	Description
inName	Text	→ Name of persistent RowSet

Discussion

This method does a deep clear of the persistent RowSet named *inName* and removes it from the list of persistent RowSets.

If no such RowSet exists, nothing happens.

columnCount**version 3**

columnCount(self) → Longint

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Longint	← Number of columns in RowSet

Discussion

This method returns the number of columns in a RowSet. Use this as the end index when iterating over the RowSet columns.

currentRow**version 3**

currentRow(self) → Longint

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Longint	← Current row of RowSet

Discussion

This method returns the current position of the cursor of *self*. If *isBeforeFirst* is *True*, this method returns *RowSet.kRow_BeforeFirst*. If *isAfterLast* is *True*, this method returns *getEnd + 1*.

dumpPersistent**version 4.0**

dumpPersistent

Discussion

This method writes a nicely formatted dump listing the name, number of rows, and column names for each persistent RowSet.

dump

version 4.0

```
dump(self {; inName {; inShowType{}}
```

Parameter	Type	Description
self	RowSet	→ RowSet reference
inName	Text	→ Optional name
inShowType	Boolean	→ True to show column type

Discussion

This method writes a nicely formatted dump of all columns and rows in *self* to the output buffer. This is very useful for debugging your queries, row callbacks and column maps.

findColumn

version 3

```
findColumn(self; inColumn) → Longint
```

Parameter	Type	Description
self	RowSet	→ RowSet reference
inColumn	Text	→ Column name to find
Result	Longint	← Index of column

Discussion

This method returns the index of a column given a name. Columns are indexed in the order they were declared on RowSet creation. If no column with the given name exists, zero is returned.

findRow

version 3

```
findRow(self; inColumn; inValue) → Longint
```

Parameter	Type	Description
self	RowSet	→ RowSet reference
inColumn	Text	→ Column in which to search
inValue	<any>	→ Value to search for
Result	Longint	← Index of first matching row

Discussion

This method performs a linear search of the column *inColumn* for the value *inValue*, starting at the first row of *self*. If a match is found, the index of the first matching row is

returned and the cursor is moved to that row. If a match is not found, zero is returned and *isAfterLast* is *True*.

The type of *inValue* must be assignment compatible with the type of the column referenced by *inColumn*.

Note: If the RowSet was created with *newFromSelection* with no subset of source rows specified, the search may be *very* slow, because each record in the selection is loaded one by one starting from the first record of the selection.

If you need to use this method with a selection-based RowSet, be sure to create the RowSet with a subset of source rows, or use a cached RowSet.

first

version 3

first(self) → Boolean

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Boolean	← True if there is a first row

Discussion

This method moves the cursor to the first row of *self*. If there is a first row (*rowCount > 0*), *True* is returned, else *False*.

getColumn

version 3

getColumn(self; inIndex) → Text

Parameter	Type	Description
self	RowSet	→ RowSet reference
inIndex	Number	→ Column index
Result	Text	← Column name

Discussion

This method returns the name of the column with the given index. Columns are indexed in the order they were declared on RowSet creation. If no column with the given index exists, an error is thrown.

getData

version 3

getData(self; inColumn) → <any>

Parameter	Type	Description
self	RowSet	→ RowSet reference
inColumn	Number/Text	→ Column index/name
Result	<any>	← Column data

Discussion

This method retrieves data from the column *inColumn* in the current row of *self*. If *inColumn* is a number, it is taken as a column index. Columns are indexed in the order they were declared on RowSet creation. If *inColumn* is a string or text, it is taken as a column name.

If *inColumn* is not a valid column index or name, an error is thrown. If *inColumn* is valid and there are no rows in the RowSet, or *isBeforeFirst* or *isAfterLast* is *True*, an empty value for the column's type is returned.

getEnd

version 4.0

getEnd(self) → Longint

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Longint	← Last row of RowSet

Discussion

This method returns the number of the last row in *self*.

getPersistentList

version 3

getPersistentList(outList)

Parameter	Type	Description
outList	String/Text Array	← Receives list of persistent RowSets

Discussion

This method retrieves a list of the persistent RowSet names into *outList*. If *outList* is not a String/Text Array, an error is generated and execution is aborted.

getRow**version 3**

getRow(self) → Longint

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Collection	← Row data

Discussion

This method returns a reference to a collection which contains all of the row data for the current row of *self*. The data in the collection is keyed on the column name. This is the easiest and most efficient way to access row data.

See "Using RowSets" on page 566 for an example of using this method to access row data.

getStart**version 4.0**

getStart(self) → Longint

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Longint	← First row of RowSet

Discussion

This method returns the number of the first row in *self*.

getTimeout**version 3**

getTimeout(self) → Number

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Number	← Timeout in minutes

Discussion

This method returns the timeout in minutes for *self*.

gotoRow

version 3

gotoRow(self; inRow) → Boolean

Parameter	Type	Description
self	RowSet	→ RowSet reference
inRow	Number	→ Row number
Result	Boolean	← True if a valid row

Discussion

This method moves the cursor to a given row within *self*.

If *inRow* is positive, the cursor is moved to the absolute row number given. If *inRow* is negative, the cursor is positioned relative to the end of the RowSet. A value of -1 positions the cursor on the last row, -2 on the second to last row, and so on.

If *inRow* > *rowCount*, this method effectively performs *afterLast*. If *inRow* is negative and *abs(inRow)* > *rowCount*, this method effectively performs *beforeFirst*.

isAfterLast

version 3

isAfterLast(self) → Boolean

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Boolean	← True if current row is after last

Discussion

This method returns *True* if the cursor is positioned after the last row of *self* or if there are no rows in *self*.

isBeforeFirst**version 3**

isBeforeFirst(self) → Boolean

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Boolean	← True if current row is before first

Discussion

This method returns *True* if the cursor is positioned before the first row of *self* or if there are no rows in *self*.

isFirst**version 3**

isFirst(self) → Boolean

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Boolean	← True if current row is first

Discussion

This method returns *True* if the cursor is positioned at the first row of *self*.

isLast**version 3**

isLast(self) → Boolean

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Boolean	← True if current row is last

Discussion

This method returns *True* if the cursor is positioned at the last row of *self*.

last**version 3**

last(self) → Boolean

self	RowSet	→	RowSet reference
Result	Boolean	←	True if there is a last row

Discussion

This method moves the cursor to the last row of *self*. If there is a last row (*rowCount* > 0), *True* is returned, else *False*.

maxRows**version 3**

maxRows(self) → Longint

Parameter	Type		Description
self	RowSet	→	RowSet reference
Result	Longint	←	Maximum rows in RowSet

Discussion

This method returns the maximum rows in *self*, as set by the *inSubset* parameter during RowSet creation.

If there is no limit on the number of rows, -1 is returned. Note that *maxRows* > *rowCount* is possible if the number of rows in the data source was less than *inSubset* at the time of RowSet creation.

move**version 3**

move(self; inRows) → Boolean

Parameter	Type		Description
self	RowSet	→	RowSet reference
inRows	Number	→	Number of rows to move
Result	Boolean	←	True if moving to a valid row

Discussion

This method moves the cursor relative to the current row of *self* by the number of rows given in *inRows*.

If $currentRow + inRows$ is in the range $1..rowCount$, *True* is returned.

If $inRows$ is positive and $currentRow + inRows > rowCount$, this method effectively performs *afterLast*. If $inRows$ is negative and $currentRow + inRows < 1$, this method effectively performs *beforeFirst*. In either case *False* is returned.

newFromArrays

version 3
modified version 4.0

`newFromArrays(inColumnMap {; inSubset {; inName {; inTimeout{}}}) → Longint`

Parameter	Type	Description
<code>inColumnMap</code>	Text	→ Virtual column mapping
<code>inSubset</code>	Number/Batch/Text	→ Maximum rows/row subset
<code>inName</code>	Text	→ Persistent name
<code>inTimeout</code>	Number	→ Timeout in minutes
Result	RowSet	← RowSet object

Discussion

This method creates a RowSet from a group of arrays.

The columns in the RowSet are determined by *inColumnMap*, which maps virtual column names to array references, which can be any valid Active4D array reference, which includes local arrays, process/interprocess arrays, and arrays within collections.

The column map is formatted as a semicolon-delimited list of *column:array* pairs. Whitespace before and after the column name and array reference is ignored.

For example, the following code would create a four-column RowSet:

```
global($ids; $names; $birthdays; $genders)
selection to array([contacts]id; $ids; \
                  [contacts]name; $names; \
                  [contacts]birthdate; $birthdays; \
                  [contacts]gender; $genders)

$map := " "
id:    $ids;
name:  $names;
birthdate:$birthdays;
gender: $genders" "

$rs := RowSet.newFromArrays($map)
```

One important thing to note in the example above is the **global** declaration of the arrays *before* they are created. For *newFromArrays* to do its magic, it must have access to any local variables that are used as array references in the column map. So any local variables used within a column map must be declared **global** before they are created or used.

Note also that *\$ids* and the other local arrays in the above example could just as easily be session items, process arrays or interprocess arrays. In that case the **global** declaration is of course unnecessary. If you need to reference an array within a collection which is local, you must declare the collection itself as **global**.

Here is another example where the session is storing a shopping cart in the parallel arrays `session{"cart.qtys"}`, `session{"cart.descriptions"}` and `session{"cart.prices"}`. To create a RowSet from the shopping cart is simple:

```
$map := ""
qty:      session{"cart.qtys"};
description: session{"cart.descriptions"};
price:    session{"cart.prices"} ""

$rs := RowSet.newFromArrays($map)
```

If all of the arrays in the column map are not the same size, the copies within the RowSet are padded to the length of the first one or the specified subset of rows, whichever is less.

For a discussion of the remaining parameters, see “newFromSelection” on page 584.

newFromCachedSelection

version 4.0

```
newFromCachedSelection(inMainTable; inColumnMap {; inSubset {; inRowCallback
  {; inName {; inTimeout}}}) → Longint
```

Parameter	Type	Description
inMainTable	Table pointer	→ Pointer to main table
inColumnMap	Text	→ Virtual column mapping
inSubset	Number/Batch/Text	→ Maximum rows/subset of rows
inRowCallback	Text	→ Code to execute when loading a row
inName	Text	→ Persistent name
inTimeout	Number	→ Timeout in minutes
Result	RowSet	← RowSet object

Discussion

This method creates a RowSet in exactly the same way as *newFromSelection*, but all of the data is preloaded into memory when the RowSet is created. For a full discussion of the parameters, see “newFromSelection” on page 584.

Reasons why you might use this method instead of *newFromSelection* include:

- You need to manipulate *inMainTable*'s selection after creating the RowSet but before it is displayed.
- You need to use the RowSet *find* method to search within the RowSet.
- A “cached” RowSet sounds a lot cooler than one that isn't.

For a discussion of the performance considerations of using cached vs. non-cached RowSets, see “To Cache or Not to Cache?” on page 588.

newFromData

version 4.0

```
newFromData(inData {; inRowDelimiter {; inColumnDelimiter {; inSubset
            {; inName {; inTimeout {; inTextFormat}}}}}) → Longint
```

Parameter	Type	Description
inData	Text/BLOB	→ Delimited data
inRowDelimiter	Text	→ Delimiter between rows
inColumnDelimiter	Text	→ Delimiter between columns
inSubset	Number/Batch/Text	→ Maximum rows/row subset
inName	Text	→ Persistent name
inTimeout	Number	→ Timeout in minutes
inTextFormat	Number	→ Storage format of text
Result	RowSet	← RowSet object

Discussion

This method creates a RowSet from delimited data, such as CSV (comma separated values), coming from a text or BLOB variable.

The first row of *inData* contains column names, followed by one or more data rows, with each row delimited by *inRowDelimiter* and each column delimited by *inColumnDelimiter*. Whitespace around the column names is trimmed.

If *inColumnDelimiter* is not passed, it defaults to “;”. If *inRowDelimiter* is not passed, it defaults to line feed (“\n”). Only the first character of each is used. If a column’s data might contain the column delimiter, you should enclose the column data in double quotes. The quotes are not included as part of the column data.

To specify a column data type, append one of the following types to the name, separated by a colon:

- boolean, date, longint, real, string(size), text

If no type is specified, the type defaults to “text”. Here is an example which creates a four-column RowSet:

```
$data := ""name:text,birthdate:date,salary:real,comments
Dewey Cheatham,06/21/1949,75000,"Great guy"
Anne Howe,03/04/1951,78000,"The Big Cheese, dude"
""
```

In the case of the string type, the number in parentheses specifies the string width.

Note: The string type in v6 is identical to the text type and has been kept for backwards compatibility.

The maximum size of the column info plus the data is 2GB.

Note that unlike the other RowSet types, if a subset of rows is specified, *sourceRowCount* will return the same as *getEnd*, because scanning of the source data is aborted as soon as the last specified row is reached. Also, row 1 is considered the first row after the header row.

If *inTextFormat* is passed and *inData* is a BLOB, it represents the storage format of the text within the BLOB. If it is not passed, the format defaults to *Mac Text without length* for backwards compatibility.

For a discussion of the remaining parameters, see “newFromSelection” on page 584.

newFromFile

version 4.0
modified v5

`newFromFile(inPath {; inRowDelimiter {; inColumnDelimiter {; inSubset {; inName {; inTimeout {; inTextFormat}}}}}) → Longint`

Parameter	Type	Description
<code>inPath</code>	Text	→ Path to file
<code>inRowDelimiter</code>	Text	→ Delimiter between rows
<code>inColumnDelimiter</code>	Text	→ Delimiter between columns
<code>inSubset</code>	Number/Batch/Text	→ Maximum rows/row subset
<code>inName</code>	Text	→ Persistent name
<code>inTimeout</code>	Number	→ Timeout in minutes
<code>inTextFormat</code>	Number	→ Storage format of text
<code>Result</code>	RowSet	← RowSet object

Discussion

This method creates a RowSet in exactly the same way as *newFromData*, but the data comes from the text file located at *inPath*.

For information on working with documents and paths, see “Working with Paths” on page 96.

newFromSelection

version 3
modified version 4.0

```
newFromSelection(inMainTable; inColumnMap {; inSubset {; inRowCallback
  {; inName {; inTimeout}}}}) → Longint
```

Parameter	Type	Description
inMainTable	Table pointer	→ Pointer to main table
inColumnMap	Text	→ Virtual column mapping
inSubset	Number/Batch/Text	→ Maximum rows/row subset
inRowCallback	Text	→ Code to execute when loading a row
inName	Text	→ Persistent name
inTimeout	Number	→ Timeout in minutes
Result	RowSet	← RowSet object

Discussion

This method creates a RowSet from a selection of records in the table pointed to by *inMainTable*.

The columns in the RowSet are determined by *inColumnMap*, which maps virtual column names to values, which can either be database fields or executable expressions. The column map is formatted as a semicolon-delimited list of *column:value* pairs. Whitespace before and after the column name and value is ignored.

If *inSubset* is a positive number, it is taken as the maximum number of rows to allow in the RowSet. In this case the start index of the RowSet will be 1 and the end index will be the number of rows in the source data or *inSubset*, whichever is less.

If *inSubset* is a negative number, this means you would like to allow an unlimited number of rows. In this case the start index of the RowSet will be 1 and the end index will be the number of rows in the source data.

If *inSubset* is a Batch collection reference, the start index of the RowSet will be *\$inSubset{"start"}* and the end index will be *\$inSubset{"end"}*.

If *inSubset* is text, it should be in the form "*<start>:<end>*". For example, to use only rows 10-20 of the source data, you would pass the string "10:20" in *inSubset*. If *inSubset* is passed in this manner, the start index of the RowSet will be *<start>* and the end index will be *<end>* or the number of rows in the source data, whichever is less.

If *inName* is passed and is non-empty, the RowSet will be persistent and will replace any previous persistent RowSet with that name.

If *inTimeout* is passed and is *> 0*, a call to *timedOut* will return true after *inTimeout* minutes. If *inTimeout* is not passed or is *<= 0*, *timedOut* will always return *False*, unless *setTimeout* is subsequently called with a value greater than zero.

Note: If you pass *inName* and create a persistent RowSet, it is effectively the same as calling *newFromCachedSelection*.

Example 1

The following code would create a four-column RowSet:

```
$map := " "
id:      [contacts]id;
name:    [contacts]name;
birthdate:[contacts]birthdate;
gender:  [contacts]gender" "

$rs := RowSet.newFromSelection(->[contacts]; $map)
```

Note the use of a literal heredoc string so the column mapping is clearer to the eye. Of course you could do this:

```
$map := "name:[contacts]name; birthdate: [contacts]birthdate; ..."
```

but it is a lot harder to see the mapping and to add/delete columns later on.

If a value references a table which is not the same as the table pointed to by *inMainTable*, Active4D will turn on auto-loading of related one records. For example, suppose we want to add a “company” column to the RowSet in Example 1, whose value is *[companies]name*. The column map would then become:

Example 2

```
$map := " "
id:      [contacts]id;
name:    [contacts]name;
birthdate: [contacts]birthdate;
gender:  [contacts]gender;
company: [companies]name" "
```

If there is an automatic relation between *[contacts]* and *[companies]*, there is nothing more that needs to be done. The RowSet will take care of loading the related *[companies]* record each time a *[contacts]* record is loaded.

But what if there is no automatic relation between *[contacts]* and *[companies]*? Ordinarily you would think of doing something like this in your web page:

Example 3

```

<%
$row := $rs->getRow

while ($rs->next)
  // have to load [companies] record
  query([companies]; [companies]id = [contacts]company_id)
%>
<tr>
  <td><%= $row{"name"} %></td>
  <td><%= $row{"birthdate"} %></td>
  <td><%= $row{"gender"} %></td>
  <td><%= $row{"company"} %></td>
</tr>
<% end while %>

```

What's wrong with this picture? You are making a direct reference to a database table within code that is meant to format output. This is a Bad Thing for two reasons:

- It breaks the insulation from hardcoded data sources that RowSets provide. If you have to write code like Example 3 you might as well not use RowSets.
- If you are using Fusebox, the code above would belong to a display fuse, which is *never* allowed to reference the database. All database references belong to query fuses.

So how can we accomplish this within the context of the RowSet? The answer is by using a *row callback*. If an expression is passed in *inRowCallback*, it will be executed just after each record in the main table is loaded, but before the column values are set.

Thus the query performed in Example 3 above would become part of the RowSet constructor call, like this:

Example 4

```

$map := ""
id:      [contacts]id;
name:    [contacts]name;
birthdate: [contacts]birthdate;
gender:  [contacts]gender;
company: [companies]name ""

$cb := "query([companies]; [companies]id = [contacts]company_id)"

$rs := RowSet.newFromSelection(->[contacts]; $map; -1; $cb)

```

Let's extend this example by adding a column "supervisor" from a third table, *[employees]*, which is not directly related to *[contacts]* at all, but whose current record is derived from some complex set of database operations that is already coded in an Active4D library method called *contacts.getSupervisor*.

In this case we will take advantage of the fact that the row callback can be a *block* of code with more than one line, since it is executed using the Active4D **execute** command. So our new version of the RowSet construction looks like this:

Example 5

```

$map := ""
id:      [contacts]id;
name:    [contacts]name;
birthdate: [contacts]birthdate;
gender:  [contacts]gender;
company: [companies]name
supervisor:[employees]name""

$cb := ""
query([companies]; [companies]id = [contacts]company_id)
contacts.getSupervisor([contacts]id)""

$rs := RowSet.newFromSelection(->[contacts]; $map; -1; $cb)

```

Note how we took advantage of heredoc strings to write multiple lines of code in the *\$callback* variable.

Now let us extend this example one more time to examine the ability to use expressions instead of fields for column values. Suppose the database is changed such that the *[contacts]name* is split into four fields: *[contacts]title*, *[contacts]first_name*, *[contacts]middle_initial* and *[contacts]last_name*. You could change the column map to include the four different fields, then concatenate the four fields in your display code, like this:

Example 6

```

$map := ""
id:      [contacts]id;
title:   [contacts]title;
firstname: [contacts]first_name;
mi:      [contacts]middle_initial;
lastname: [contacts]last_name;
birthdate: [contacts]birthdate;
gender:  [contacts]gender;
company: [companies]name
supervisor:[employees]name""

// display code
<td><%=concat(" "; $row{"title"}; $row{"firstname"}; \\
              $row{"mi"}; $row{"lastname"}) %></td>

```

While you *could* do this, you shouldn't, because you are effectively tying yourself to the database structure in your display code. What you *really* want is a virtual database field that concatenates the four fields together. You can accomplish this easily by using an expression as the column value in the column map.

To use an expression as the column value, enclose any valid Active4D expression which returns a value in a pair of backticks (`). The expression will be evaluated after each record is loaded and the row callback is executed.

This allows us to change the column map and display code in Example 6 to the following:

Example 7

```

$map := ""
id:      [contacts]id;
name:    `concat(" "; [contacts]title; [contacts]first_name;
contacts]middle_initial; [contacts]last_name)`;
birthdate: [contacts]birthdate;
gender:   [contacts]gender;
company:  [companies]name
supervisor:[employees]name" "

// display code
<td><%= $row{"name"} %></td>

```

Note that the display code is the same as in Example 3, which means it did not have to change, even though a huge change was made to the database structure!

Together with row callbacks, column expressions give you unparalleled power to isolate the database from the rest of your code. And since any valid Active4D expression can be used, you can call a method as well. Overall selection-based RowSets give you much of the expressive power of SQL.

Note: Unlike row callbacks, column expressions may only be one line of code. If you need to execute multiple lines of code, put them in a method and call the method.

To Cache or Not to Cache?

There are two types of selection-based RowSets: cached and non-cached. This method creates a non-cached RowSet, which means that every time you go to a row a record has to be loaded from the database, the row callback has to be executed, column data has to be copied from the record, and column expressions have to be executed. The method *newFromCachedSelection* creates a cached RowSet, which means all of the row data is preloaded into memory when the RowSet is created, and all of the above mentioned operations only occur once.

There are several factors which might influence the decision whether or not to use cached RowSets:

- If the number of rows and columns is very large you may have to worry about memory usage, however the memory is freed as soon as the request is processed.
- If you will be accessing the RowSet multiple times — for example if you need to do a search within the RowSet before displaying it — then a cached RowSet will be faster, since loading data from an array is faster than loading a record and accessing its fields, especially if Active4D is running on 4D Remote.
- If there is no row callback and no column expressions, a cached RowSet will use **SELECTION TO ARRAY** to load the data. If Active4D is running on 4D Remote, and if the number of columns used in the RowSet is a small subset of the fields in the source table, or the source table contains large text, BLOB or picture fields, then **SELECTION TO ARRAY** is faster than a row-by-row loading of records because only data from the requested fields is sent to the 4D Remote.

next**version 3**

next(self) → Boolean

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Boolean	← True if there is a next row

Discussion

This method moves the RowSet cursor to the next row in the RowSet and loads its data. If the cursor was not the last row of the RowSet before this call, *True* is returned, else *False*.

persistent**version 4.0**

persistent(inName) → Collection

Parameter	Type	Description
inName	Text	→ Name of a persistent RowSet
Result	RowSet	← Named RowSet

Discussion

This method returns a reference to the persistent RowSet named *inName*. If no such persistent RowSet exists, zero is returned.

previous**version 3**

previous(self) → Boolean

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Boolean	← True if there is a previous row

Discussion

This method moves the RowSet cursor to the previous row in the RowSet and loads its data. If the cursor was not the first row of the RowSet before this call, *True* is returned, else *False*.

rowCount**version 3**

rowCount(self) → Longint

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Longint	← Number of rows in RowSet

Discussion

This method returns the number of rows in the RowSet. To get the number of rows in the data source on which the RowSet is based, use *sourceRowCount*.

setColumnArray**version 3**

setColumnArray(self; inColumn; inArray)

Parameter	Type	Description
self	RowSet	→ RowSet reference
inColumn	Text	→ Virtual column to set
inArray	Array	→ Column data

Discussion

If the RowSet *self* does not already have a column named *inColumn*, a new column will be added with the given name and data. If a column already exists with the same name, its data is replaced.

If the size of *inArray* is less than *rowCount*, it is expanded accordingly. If the size of *inArray* is larger than *rowCount*, any other columns are expanded accordingly. If *maxRows* ≥ 0 , the size of the columns is clipped to that.

This method is designed to be used in cases where the bulk of your data is non-calculated but one column needs to be calculated.

setColumnData**version 3**

```
setColumnData(self; inColumn; inData {; inDelimiter;})
```

Parameter	Type	Description
self	RowSet	→ RowSet reference
inColumn	Text	→ Virtual column to set
inData	Array	→ Column data
inDelimiter	Text	→ Row delimiter

Discussion

This method functions like *setColumnArray*, but the column data comes from delimited data instead of an array. If *inDelimiter* is not passed, it defaults to “;”.

setRelateOne**version 3
modified version 4.0**

```
setRelateOne(self; inRelateOne) → Boolean
```

Parameter	Type	Description
self	RowSet	→ RowSet reference
inRelateOne	Boolean	→ True to load related one records
Result	Boolean	← Old value of flag

Discussion

This method sets the auto relate one flag, which determines whether related one records will be loaded when moving the cursor in a RowSet that was created with *newFromSelection*.

The old value of the flag is returned.

setTimeout

version 4.0

```
setTimeout(self; inTimeout)
```

Parameter	Type	Description
self	RowSet	→ RowSet reference
inTimeout	Number	→ Timeout in minutes

Discussion

This method sets the timeout of *self* to *inTimeout* minutes after *self*'s creation. In most cases this method is unnecessary, as you can pass the timeout when you create a RowSet.

sort

version 4.0

```
sort(self; inSortMap)
```

Parameter	Type	Description
self	RowSet	→ RowSet reference
inSortMap	Text	→ Specifies columns to sort and sort order

Discussion

This method sorts the data in *self* according to the sort specification given in *inSortMap*. The RowSet must have been created with:

- RowSet.newFromCachedSelection
- RowSet.newFromArrays
- RowSet.newFromData
- RowSet.newFromFile

inSortMap is a semicolon-delimited list of one or more *column:order* pairs, where *column* is the name specified in the original column map when the RowSet was created, and *order* is one of the comparison operators $<$ $>$ $=$. You can perform a multi-level sort by passing more than one column with an order of $<$ or $>$.

If *order* is $<$, the column is sorted in descending order. If *order* is $>$, the column is sorted in ascending order. If *order* is $=$, the column is sorted neutrally along with the previous column in the list.

Any columns in the RowSet that are not included in *inSortMap* will be sorted neutrally along with the other columns.

Example

```
$map := "lastname:<;firstname:>"
$rowset->sort($map)

// RowSet will be sorted by lastname descending
// and firstname ascending.
```

sourceRowCount**version 3**

sourceRowCount(self) → Longint

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Longint	← Number of rows in RowSet data source

Discussion

This method returns the number of rows in the data source on which the RowSet is based. To get the number of rows in the RowSet itself, use *rowCount*.

Note: If the RowSet was created with *newFromData/newFromFile* and a subset of rows, this method will return the same as *getEnd*.

timedOut**version 4.0**

timedOut(self) → Boolean

Parameter	Type	Description
self	RowSet	→ RowSet reference
Result	Boolean	← True if RowSet has timed out

Discussion

If *self* has a timeout value ≤ 0 , this method returns *False*.

Otherwise this method calculates the difference between the current timestamp and the creation timestamp of *self*, and if the result is greater than *self*'s timeout, *True* is returned, else *False*.

SessionHandler

The SessionHandler library provides a ready to use session handler that stores sessions in the database. This is necessary when you have multiple servers behind a load balancer, because there is no guarantee any given server will serve all requests from a given user once a session starts.

The methods in the SessionHandler library are documented here in case you want to create your own session handler, either in 4D code or in a library, and to help you debug any problems that might arise when using this library or the predefined 4D method-based session handler.

nextId**v6.2**

nextId → Longint

Parameter	Type	Description
Result	Longint	← Next internal session id

Discussion

This method is mandatory, and is responsible for returning the next available internal session id. This method is called when a new session is created.

If zero is returned, any attempt to work with the session will result in an error being thrown and execution being aborted.

read**v6.2**

read(id) → BLOB

Parameter	Type	Description
id	Longint	→ Internal session id
Result	BLOB	← Session data

Discussion

This method is mandatory, and is responsible for returning the session data for the session with the given id.

If no such session exists, an empty BLOB should be returned.

If a matching session is found, this method should wait until it can obtain a lock on the session's storage and leave it in a locked state so that no other processes can access the session until the current process has finished.

This method is called every time a user makes a request that contains a session id cookie, query parameter, or form field.

write**v6.2**

write(id; data; expires) → Boolean

Parameter	Type	Description
id	Longint	→ Internal session id
data	BLOB	→ Session data
expires	Real	→ Time when session will expire
Result	Boolean	← <i>True</i> for success

Discussion

This method is mandatory, and is responsible for saving the *id*, *data* and *expires* values. If existing storage with the given id exists, it should be updated. In a library method, the *data* parameter should be a reference parameter (prefixed with "&") which avoids an extra copy of the BLOB.

Note: *expires* is a datetime value that cannot be manipulated by 4D.

When finished, this method should release the lock on the session storage that was obtained with the **read** method.

This method is called at the end of every request in which **nextid** or **read** is called. If *False* is returned, the request fails and returns 500 (Internal server error) as the status code.

delete**v6.2**

delete(id) → Boolean

Parameter	Type	Description
id	Longint	→ Internal session id
Result	Boolean	← <i>True</i> for success

Discussion

This method is optional, and is responsible for deleting storage for the session with the given id and releasing any locks on it.

This method is called at the end of a request in which **abandon session** has been called. If *False* is returned, the request fails and returns 500 (Internal server error) as the status code.

If this method is not implemented, be sure to implement some other mechanism for cleaning up deleted and expired sessions.

purge**v6.2**

purge(maxExpires) → Boolean

Parameter	Type	Description
maxExpires	Real	→ Maximum valid expires value
Result	Boolean	← <i>True</i> for success

Discussion

This method is optional, and is responsible for deleting storage for all session whose expires time is less than *maxExpires*.

This method is called by the Active4D housekeeper at the interval defined by the "session purge interval" configuration option. If *False* is returned, an error is logged in the Active4D log.

If this method is not implemented, be sure to implement some other mechanism for cleaning up deleted and expired sessions.

CHAPTER 13

Debugging

Although Active4D does not have an interactive debugger, there are many tools you can use to find out what is going on inside your code. This requires putting some extra debug code in your scripts, but such is life.

The Basics

There are several basic techniques for peeking inside your code to see what is going on. The technique you use will depend on what the code is doing and whether it is possible to write debugging information to the page.

Using write

The simplest way to find out the internal state of Active4D is to use one of the **write** commands to write debugging information to your page, like this:

```
for ($i; 1; records in selection([Ingredients]))
  goto selected record([Ingredients]; true)
  writebr([Vendors]Name) // debug code
  // normal code
end for
```

Tracing execution

In some cases you will come across an error in your code which you cannot explain. In such cases, often the line that caused the error is expecting some condition that was not fulfilled, and the actual bug is in code that came earlier.

For example, consider the following code:

```
$c := new collection
$c{"foo"} := "bar"
include("bar.a4d")

// in bar.a4d
writebr("%d-%d-%d" % (year of($c{"bar"}); \\  
                    month of($c{"bar"}); \\  
                    day of($c{"bar"})))
// above line fails, $c{"bar"} not set
```

The error message you receive will be that Active4D was expecting a date in the expression *year of(\$c{"bar"})*. Let's pretend there is a lot of code and HTML between the

creation of `$c` and the **write** statement in `bar.a4d`, and you are sure something in there was supposed to set `$c("bar")`. Obviously it isn't happening, so you have to work your way back from the line that caused the error and figure out what went wrong.

There are two strategies you can take:

- Insert **write to console** commands at strategic points, allowing execution to continue to the line that caused the error.
- Dump debugging information and stopping execution before the error occurs.

Note that in the second case, you must stop execution, otherwise the debugging information you dump will be replaced by the error message when the error occurs. To stop execution, you would do something like this in a Fusebox fuse or layout:

```
a4d.debug.dump collection($c; ""); true)
return
```

Why **return**? Because **return** will halt execution only of the current include file. If you halt execution entirely, the Fusebox core will not have a chance to continue execution to the point where your debug output is actually written to the response buffer.

If you are not using Fusebox, you would do something like this:

```
a4d.debug.dump collection($c; ""); true)
exit
```

In this case you want to use the **exit** command, which halts all execution.

Standard Library Methods

Active4D provides a wealth of library routines for dumping the current state of the interpreter to the client browser. You will find them to be invaluable debugging tools.

a4d.console and **a4d.debug**

The *a4d.debug* library provides many handy methods for dumping information about arrays, collections, RowSets, and selections, as well as other internal information.

There are times, however, when you cannot dump debugging information onto a page. In such cases the *a4d.console* library provides similar functionality to the *a4d.debug* library, but dumps its information to the Active4D debugging console.

Typically you would place a call to methods in these libraries in a script where you want to view a snapshot of the relevant information.

For example, to view the contents of the current session, you only need one line of code:

```
<% a4d.debug.dump session %>
```

For details on the *a4d.debug* methods, see “a4d.debug” on page 437. For details on the *a4d.console* methods, see “a4d.console” on page 432.

The Active4D Log

Active4D v6 has extensive logging of its internal operations to help you debug problems that are difficult to trace otherwise.

Logs are kept in <database structure directory>/Logs/Active4D, where the “Logs” directory is what would be returned by **Get 4D folder**(*Logs Folder*). Log files are rotated automatically when they reach 1MB in size. A total of seven log files are kept, with Active4D.0.log being the current log file, Active4D.1.log being the previous log file, and so on up to Active4D.6.log.

Active4D logs the following types of information in the log file:

- Information about the host environment
- Loading of configuration files
- Loading and unloading of libraries
- License key file information
- Internal and runtime errors

Each log entry occupies one logical line and looks something like this:

```
Oct 20 09:37:57 Active4D: [info] license: matched IP address:
192.168.1.7
```

Log entries contain the date and time of the entry, followed by “Active4D:”, followed by the entry type, followed by the message. For non-runtime error messages, the message is prefixed with the module within Active4D where the message originated. For example, the entry above originated in the “license” module.

When a runtime error occurs, the log message contains the error message, followed by the location of the error. Here is a typical syntax error log entry:

```
Oct 20 19:43:05 Active4D: [error] Syntax error:
/Users/aparajita/Active4D/demo.4dbase/web/dsp_basics.a4d, line 22
```

The log entry types are:

- **info**: General information about Active4D operations or environment, as well as regular messages logged with the **log message** command
- **notice**: “Official” announcements
- **warn**: Conditions that may cause problems or errors and should be looked into
- **error**: Internal or runtime errors that should be attended to, and error messages logged with the **log message** command
- **debug**: Detailed information about Active4D’s internal operations

Changing the Log Level

If the normal logging does not provide enough information to debug a problem, or if you would like to disable logging altogether, you can change the log level.

To change the log level, follow these steps:

- 1 In a text editor, create a new plain text document.
- 2 In the document, enter the text “debug” or “off”.
- 3 Save the document as “log_level” in the Active4D log directory.
- 4 Restart 4D.

If Active4D finds “log_level” (or for backward compatibility, “log_debug_level”) in the log directory and it contains “debug” or “off”, the log level is set accordingly.

- When the log level is “debug”, you will see many log extra entries of type “debug”. This level gives you detailed information about the inner workings of Active4D.
- When the log level is “off”, logging is completely turned off.

The default log level can be restored either by moving, renaming or deleting the “log_level” file or by deleting the text within the file, then restarting 4D.

The Session Editor

One of the greatest strengths of Active4D is its session management, which you will want to use heavily. Since much of your code will be working with sessions, you need tools to see what is happening inside the session.

Active4D provides two tools for debugging sessions: a client-side session editor, and a server-side session monitor.

Using the Session Editor

Once a session has been created on the server, you can use the client-side session editor to view and edit the session. The session editor is completely contained in the Active4D script "sed.a4d". To use the session editor, follow these steps:

- 1 Place "sed.a4d" somewhere within your web root folder (or a subfolder thereof).
- 2 If you are using Fusebox, add a fuseaction in some circuit's fbx_switch.a4d that will include "sed.a4d".
- 3 Open a new window or tab in your browser. If you are on Windows, this means creating a new browser window or tab from within the browser, not launching a new browser. Otherwise you will not be sharing the same session cookie.
- 4 Enter the URL of "sed.a4d" or of the fuseaction that will ultimately include it.

Once the session editor appears, you will see a screen something like this:

Session [2320445/A1CABAA0F6E32220]		
Timeout: 15 minutes		
Key	Type	Value
bar	Longint	7
foo	Unicode String	"bar"

The upper part of the session editor contains the fields and controls you use to edit the session. The lower part of the session editor displays the contents of the current session.

The elements of the session editor are:

- **Item key:** Enter the name of the session item you wish to set or get the value of.
- **Value type:** When setting an item, this popup determines what type of value will be set. You may change the type of an existing session item by using this popup.

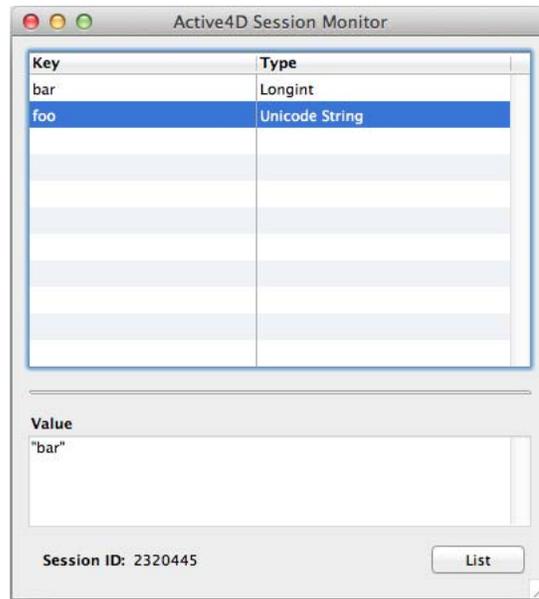
- **Array index:** If you are setting or getting an element of an array item, specify its index here.
- **Value:** When setting an item, you enter the value here. The text you enter is converted according to the type set in the *Value type* popup menu. If you are setting an array, you can enter multiple items. For example, to set a Longint item array with four elements, select "Array Longint" from the *Value type* popup menu and enter "1 2 3 4" in this field.
- **Array separator:** By default, a space is the array separator used to parse array elements in the *Value* field. If you want to use another separator, enter the character in this field. For example, to set a text array with two elements, "O give me" and "a home", you would enter ";" in the *Array separator* field and the text "O give me,a home" in the *Value* field.
- **Set button:** Use this button to set an item with the values you have specified. If no item with the specified item key exists, a new item will be added to the current session with the given key and value.
- **Get button:** Use this button to retrieve the value of the specified item or item array element.
- **Delete button:** Use this button to delete the item specified by *Item key*.
- **Abandon button:** Use this button to abandon the session entirely. After clicking this button the display may not change, but the session has been abandoned. To update the display, click the *Get* button after clicking *Abandon*.
- **Set Timeout button:** The current timeout (in minutes) is always displayed in the field to the left of this button. To change the timeout, enter a new number (fractional minutes are allowed) in the field and click the button.

The Session Monitor

While the Session Editor is ideal for viewing and editing a single session during development, once your web site is deployed you may want to see all of the current sessions. This is what the Session Monitor is for.

Displaying the Session Monitor

If you are using the Active4D shell, or have installed the Active4D Insider group, the Session Monitor is available via the Active4D menu. Once opened, the Session Monitor looks something like this:



The upper part of the session data view displays all of the items in the session. The lower part of the session data view displays a textual representation of the selected item. Arrays are displayed as a CR-delimited list of elements.

Clicking on an item in the upper list displays its value below. To return to the session list, click the *List* button or press Return.

The Active4D Debugging Console

Sometimes you can't display a dump on the client browser because an error is occurring which replaces the dump with an error message. In other cases it may be necessary to print out some messages while execution is proceeding so you can trace the internal state.

Active4D provides a server-side debugging console which can be written to during execution. By default the console is disabled in a compiled database. To enable the debugging console in a compiled database, do the following:

- 1 Open the method `A4D_DebugInitHook`.
- 2 Uncomment the line `^$ioDebug->:=True`, or if no such line exists, add such a line (uncommented).
- 3 Close the method and recompile your structure.

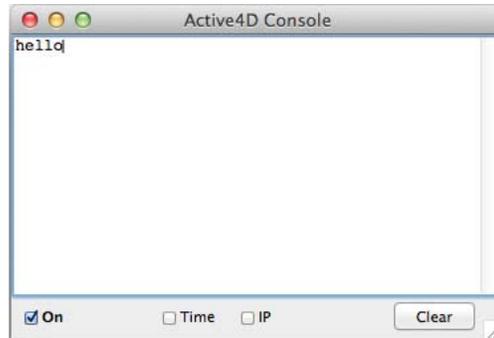
Using the Debugging Console

To write debugging info to the server-side console, simply use the **write to console** command, which takes an expression of any type and converts it to text.

This text is then appended to the end of the console's output. If the console was not previously visible, it is shown and brought to the front. Active4D takes care of showing

the output from each **write to console** call on a new line; you do not have to add a carriage return at the end yourself.

Here is what the console looks like:



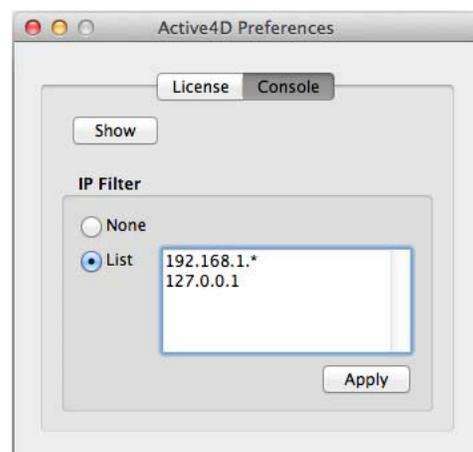
You can turn the console on or off globally by clicking on the *On* checkbox. In addition, you can have console messages prefixed by the time and/or IP address of the source by checking the *Time* and *IP* checkboxes.

The console log is automatically cleared when it reaches 5K in size. If you want to clear it before then, click the *Clear* button.

The debugging console works both with 4D standalone and with Client/Server. When run under 4D Remote with 4D Server acting as the web server, console messages are queued on Server and retrieved by the 4D Remotes on a first come, first serve basis. Each 4D Remote that has the console on checks the Server once per second to see if any console messages are queued. If so, all of the queued messages are copied and removed from the queue. Thus only one workstation may view any queued console messages.

Filtering Console Messages

You may configure the console to listen to messages only from certain IP addresses. To do so, run the *A4D_Prefs* method (either directly or through a menu item) and click the *Console* tab in the dialog. You will see a dialog something like this:



The *IP Filter* section determines which messages are displayed in the console. If the *None* radio button is selected, all console messages will be displayed. If you select the *List* radio button, you may enter a carriage return-delimited list of IP addresses to listen to. To make the filter list active, click the *Apply* button.

For example, the following list would cause the console to listen to only two IP addresses:

```
192.168.1.7  
192.168.1.13
```

You may also use a * at the end of an IP address to indicate that any address within a subnet is valid. For example, the address:

```
192.168.1.*
```

would allow the console to listen to any IP address within the range *192.168.1.0* through *192.168.1.255*.

When you close the Preferences dialog, your filter list is saved to a preferences file on your machine and is read back in the next time you launch the shell.

Tuning Console Refresh

If you are using the console on 4D Remote when 4D Server is the web server, the console has to communicate with the server (using **GET PROCESS VARIABLE**) to get the latest console messages. Unfortunately, on Windows **GET PROCESS VARIABLE** causes CPU usage to momentarily go over 30%. To reduce the impact of this, by default the console only checks the server every 3 seconds.

You can change the refresh delay as follows:

- Open the A4D_Config list in the Tool Box.
- Look for the item "ConsoleRefreshDelay=N", where N is a number.
- Change the number to the desired number of seconds between console refreshes.
- Close the Tool Box and restart 4D.

CHAPTER 14

Error Handling

Active4D goes to considerable lengths to catch errors and display meaningful error messages. If any errors occur during the execution of an Active4D program, execution is immediately aborted and the error handler takes over. The error handler will do one of two things:

- Display a default error message.
- Attempt to execute the current error page.

Note: This chapter describes execution error handling. For a description of http error handling, see “HTTP Error Handling” on page 63.

The Default Error Message

The default error message is designed to be a debugging aid. Using the information in the error message, you can quickly determine the source of the error.

Here is what an error message looks like:

Active4D error		
Source	Method	Line
/Users/Shared/Active4D-demo.4dbase/web/index.a4d	[main]	12
[text block]	fusebox.core	117
/Users/Shared/Active4D-demo.4dbase/web/techniques/errors/fbx_switch.a4d	[main]	28
/Users/Shared/Active4D-demo.4dbase/web/techniques/errors/controllers/act_executionError.a4d	[main]	15
\$foo := \$bar // this will cause an error		
Reference to an undefined variable or value		
Active4D 6.0b1 [Macintosh/Intel, release, 32-bit]		

First Active4D shows you the call chain, starting from the least recent at the top to the most recent at the bottom. Included files are all listed separately as part of the chain.

After the call chain, the offending line of code is displayed, followed with a message describing what went wrong. The error-causing line is displayed with the following color coding:

- **blue:** Tokens already evaluated.
- **red:** The offending token or the last token in the offending expression.
- **gray:** Tokens not yet evaluated.

This makes it easy to find the exact cause of the error.

Note: The line number for source files is the absolute line number within the file, whereas the line number for a method is the line number of the body of that method, not of the file containing the method. Since empty lines are stripped out of methods, the line number does not include empty lines.

Using a Custom Error Page

The default error message is fine when you are developing a web site, but for public consumption it is not exactly user-friendly.

Active4D allows you to execute another Active4D script file when an execution error occurs. This allows you to display a user-friendly message and maintain the look and feel of your web site. Within an error handler, the response status is initially *HTTP Status OK* (200). You can retrieve the error status that triggered the handler with the **get error status** command.

The error page is not static HTML, but a fully dynamic Active4D file with full access to all of Active4D's features. Thus your error page could log error information to the database, if you so choose.

Note: A custom error page is executed within the context of the page that caused the error, as if it were included at the point at which the error occurred.

You can dynamically set the file to use as the error page in two ways:

- Set the "error page" option in Active4D.ini
- Call **set error page** within Active4D

If the designated error page cannot be found, is in a forbidden directory, or an error occurs during the execution of the error page, Active4D falls back to the default error message.

The "error page" Option

The "error page" option in Active4D.ini allows you to specify a URL-style path to a file to execute when an error occurs. The path is relative to the current root.

If you set an error page in Active4D.ini, all Active4D sessions will have their error page set by default to the page you specify. In a production system you will most likely want to do this to establish a "fail-safe" error page in case all else fails.

The "set error page" Command

Within an Active4D session, you may want to change the error page depending on the current context. In this case you can use the **set error page** command within your Active4D code.

The **set error page** command takes a Text parameter which is the same type of path you use for the “error page” option in Active4D.ini. To get the current error page, should you wish to save and restore the current error page setting, use the **get error page** command.

Error Page Variables

Within the context of an error page execution, there are several local variables defined for you to help you identify the source of the problem within your error page:

Name	Description
\$a4d_err_type	A general classification of the error
\$a4d_err_msg	A description of what went wrong
\$a4d_err_source_path	The path to the file being executed, if any
\$a4d_err_source_method	The library and method name being executed, if any
\$a4d_err_before_token	The portion of the line before the token which caused the error
\$a4d_err_token	The token which caused the error (or the closest guess)
\$a4d_err_after_token	The remainder of the line after the offending token
\$a4d_err_line_num	The line number being executed at the time of the error
\$a4d_err_code	The throw code if throw caused the error

All of the above variables are Text except for *\$a4d_err_line_num*, *\$a4d_err_type*, and *\$a4d_err_code*, which are Longints.

The values used by *\$a4d_err_type* have 4D constants defined so you can test against them in your code. The error types are:

Type	Description
A4D Error Type Unexpected	An error occurred that could not be classified by Active4D
A4D Error Type Syntax	A syntax error was encountered in your code
A4D Error Type Database	A database engine error occurred during the execution of a database command
A4D Error Type IO	An I/O error occurred during the execution of a document command or a file could not be found
A4D Error Type Critical	A potentially fatal condition has occurred such as running out of memory
A4D Error Type Runtime	A runtime error such as a bad array index has occurred

You may wish to use common Active4D code that will do something different in the context of an error page. You can do this by using the **in error** command. Within the

context of an error page execution this command returns the Boolean value *True*, otherwise *False*.

If an error handler is executed because of a **throw** statement, you can retrieve the throw code and message with the **throw code** and **throw message** commands.

Custom Error Handling in Fusebox

If an error occurs during the execution of a Fusebox application, the Fusebox core will not finish executing. If you have set a custom error page, that page will be executed but it will not be within the context of Fusebox.

Since you really want everything within a Fusebox application to operate within the Fusebox context, what you really want is to execute a fuseaction when an error occurs. To facilitate this, the *fusebox* library includes a special method for handling execution errors.

fusebox.handleErrorInline puts all of the *\$a4d_err* variables listed above into *\$attributes* as items with the same name, but without the '\$' prefix. In addition, the following items are added to *\$attributes*:

Name	Description
a4d_err_url	The full requested URL that led to the error
a4d_err_referer	The HTTP referer of the original request
a4d_in_handler	Boolean True

Assume we have set the "error page" option in Active4D.ini to "act_handleError.a4d". Here is what *act_handleError.a4d* should look like:

```
<% fusebox->handleErrorInline("home.handleError") %>
```

Of course you may change "home.handleError" to whatever fuseaction you wish. The *fbx_switch.a4d* of our home circuit would have a case like this:

```
case of
  :($fusebox{"fuseaction"} = "handleError"
    include("dsp_handleError.a4d")
```

The *dsp_handleError.a4d* display fuse would hopefully display some meaningful message and perhaps log the error that occurred.

Within a fusebox error handler, the "fusebox page" option is temporarily ignored, which allows you to execute non-fusebox scripts if you wish.

Note: Active4D v4.x used the method **fusebox.handleError** for handling errors. This method still works but has been deprecated in favor of **fusebox.handleErrorInline**, as the new method is executed without redirecting.

When fusebox itself throws an error, for example if it is given an invalid circuit name, it executes **throw** with one of the following throw codes, which are defined as fusebox library constants:

Code	Description
<code>fusebox.kMissingCircuitDefinitionsError</code>	<code>fbx_circuits.a4d</code> could not be found
<code>fusebox.kEmptyCircuitDefinitionsError</code>	<code>fbx_circuits.a4d</code> was found but no circuits were defined
<code>fusebox.kMissingRootSettingsError</code>	The root <code>fbx_settings.a4d</code> could not be found
<code>fusebox.kMissingFuseactionError</code>	<code>fusebox.conf.fuseaction</code> is not defined
<code>fusebox.kInvalidCircuitError</code>	The requested circuit cannot be found in <code>fbx_circuits.a4d</code>
<code>fusebox.kMissingSwitchError</code>	<code>fbx_switch.a4d</code> is missing in the target circuit directory
<code>fusebox.kInvalidActionError</code>	Set by the <code>fusebox.invalidAction</code> method
<code>fusebox.kMissingLayoutError</code>	The layout file specified in <code>fbx_layouts.a4d</code> cannot be found

The Active4D Log

Active4D can log errors to a file called “Active4D.log”, located in a directory called “Active4D logs” within the log directory. For information on the log directory, see “The Active4D Log” on page 601.

Each log entry contains the date and time of the error, the severity of the error, where the error occurred, and what went wrong.

You can determine which types of errors to log, or turn logging off altogether, by using setting the “log level” option in Active4D.ini or by using the **set log level** command within Active4D.

Setting the “log level” option sets the default log level for all future Active4D sessions. Calling **set log level** within Active4D sets the log level dynamically within a single Active4D execution.

Log Levels

There are four error log levels, each with an associated named constant. Each level has an associated bit flag.

Named constant	Value	Description
A4D Log Off	0	No messages are logged
A4D Log Critical Errors	1	Fatal runtime errors like out of memory

Named constant	Value	Description
A4D Log User Messages	2	User message logged through the log message command
A4D Log Execution Errors	4	Execution errors such as syntax errors

You can set exactly which types of errors are logged by adding the bit flags together and using that value. If you are setting the log level within Active4D code, you can use the named constants and also use the bitwise operators `|` (OR) or `^` (XOR) to set or clear one flag. To turn logging off completely, set the log level to zero.

The default log level is 7, which logs all messages. You can retrieve the current log level within Active4D with the **get log level** command.

APPENDIX A

Index of Commands

_form	202
_query	271
_request	330
{ (appending index)	142
{-<index>} (from end index)	142
% (formatting operator)	363
%% (formatting operator)	364
=	311
A4D Abandon session	419
A4D Base64 decode	419
A4D Base64 encode	419
A4D Blowfish decrypt	420
A4D Blowfish encrypt	420
A4D Execute <type> request	66
A4D Execute 4D request	70
A4D Execute BLOB request	69
A4D Execute BLOB	73
A4D Execute file	72
A4D Execute stream request	70
A4D Execute text	73
A4D FLUSH LIBRARY	420
A4D Get IP address	421
A4D GET LICENSE INFO	424
A4D Get MAC address	422
A4D Get MAC address	424
A4D Get root	422
A4D GET SESSION DATA	422
A4D GET SESSION STATS	423
A4D Get time remaining	424
A4D Get version	425
A4D Import library	425
A4D LOG MESSAGE	425
A4D MD5	426
A4D Native to URL path	426
A4D RESTART SERVER	426
A4D Set HTTP body callback	427
A4D SET ROOT	427
A4D STRIP 4D TAGS	428
A4D URL decode path	428

A4D URL decode query.....	428
A4D URL encode path.....	429
A4D URL encode query.....	429
A4D URL to native path.....	429
abandon response cookie.....	319
abandon session.....	357
add datetime to json.....	222
add element.....	143
add function to json.....	223
add rowset to json.....	224
add selection to json.....	226
add to json.....	220
add to timestamp.....	404
add.....	446
add.....	545
addArray.....	449
addCSS.....	558
addDateTime.....	450
addDumpStyles.....	559
addFunction.....	451
addJavascript.....	560
addJS.....	560
addMetaTag.....	562
addRowSet.....	452
addSelection.....	455
afterLast.....	571
ALL RECORDS, FIRST/LAST/NEXT/PREVIOUS RECORD ...	340
Append document.....	389
append to array.....	143
append.....	471
applyToSelection.....	482
ARRAY <type>.....	144
arrayToList.....	471
articleFor.....	482
auth password.....	411
auth type.....	411
auth user.....	411
authenticate.....	412
auto relate.....	340
base64 decode.....	169
base64 encode.....	170
beforeFirst.....	571
blob to collection.....	156
blob to session.....	351
blobToCollection.....	483
blobToSession.....	483
blowfish decrypt.....	171
blowfish encrypt.....	172

br.....	509
buffer size.....	301
build query string.....	275
buildOptionsFromArrays.....	509
buildOptionsFromLists.....	510
buildOptionsFromOptionArray.....	511
buildOptionsFromOptionList.....	512
buildOptionsFromRowSet.....	513
buildOptionsFromSelection.....	514
call 4d method.....	239
call method.....	239
camelCaseText.....	484
capitalize.....	365
cell.....	366
changeDelims.....	472
checkboxState.....	515
checkSession.....	515
choose.....	240
chopText.....	484
clear array.....	145
clear buffer.....	301
clear collection.....	160
clear object.....	259
clear response buffer.....	301
clear.....	433
clearPersistent.....	571
collection has.....	166
collection to blob.....	156
collection to object.....	259
collection.....	154
collectionItemsToQuery.....	516
collectionToBlob.....	485
collectionToQuery.....	517
columnCount.....	572
compare strings.....	367
concat.....	368
configuration.....	333
contains.....	472
containsNoCase.....	473
convertJSONDates.....	468
COPY ARRAY.....	145
copy collection.....	158
copy upload.....	196
core.....	551
count collection items.....	167
count form variables.....	205
count globals.....	212
Count in array.....	146

count query params	275
count request cookies	291
count request infos	294
count response cookies	319
count response headers	322
count session items	356
count uploads	197
Create document	389
cud	486
current file	390
current library name	187
current line number	187
current method name	187
current path	390
current platform	330
current realm	412
currentRow	572
Date	182
day of year	182
deep clear collection	160
deep copy collection	158
default directory	391
define	242
defined	414
delete collection item	167
DELETE FOLDER	391
delete global	213
delete response cookie	319
delete response header	323
delete session item	356
Delete string	368
delete	596
deleteAt	473
deleteSelection	491
directory exists	392
directory of	392
directory separator	392
dump array	433
dump array	438
dump collection	434
dump collection	439
dump configuration	440
dump form variables	435
dump form variables	440
dump license info	435
dump license info	440
dump locals	441
dump query params	435

dump query params	441
dump request info	436
dump request info	442
dump request	442
dump selection	442
dump session stats	444
dump session	436
dump session	443
dump	573
dumpDefaults	532
dumpLib	545
dumpPersistent	572
embedCollection	517
embedCollectionItems	519
embedFormVariableList	519
embedFormVariables	519
embedQueryParams	520
embedVariables	520
emptyTag	521
enclose	369
encode	463
encodeArray	463
encodeBoolean	464
encodeCollection	464
encodeDate	466
encodeString	466
end json array	231
end json object	232
end save output	304
endArray	459
endObject	460
execute in 4d	244
EXECUTE	244
extension of	393
Field name	176
file exists	393
filename of	393
fill array	146
filterCollection	492
find	473
findColumn	573
findNoCase	474
findRow	573
first not of	370
first of	371
first	474
first	574
for each/end for each	216

for each/end for each.....	245
form variables has	202
form variables	202
format string.....	371
formatUSPhone.....	492
formVariableListToQuery.....	521
full requested url	330
fuseboxNew	546
get auto relations.....	341
get cache control.....	325
get call chain.....	188
get collection array size	162
get collection array	161
get collection item count.....	163
get collection item	163
get collection keys.....	164
get collection	161
get content charset.....	327
get content type.....	327
get current script timeout	336
get error page	191
get error status	191
get expires date	326
get expires	325
get field numbers.....	176
get field pointer	177
get form variable choices	203
get form variable count	204
get form variable	203
get form variables	205
get global array size	210
get global array.....	209
get global item	210
get global keys	211
get global.....	208
get http error page	192
Get indexed string.....	298
get item array.....	218
get item key	217
get item type	217
get item value	217
get license info	331
get local.....	414
get log level.....	192
get output charset.....	305
get output encoding	306
get platform charset.....	335
Get pointer	247

get query param choices	272
get query param count	273
get query param	272
get query params	274
get request cookie	290
get request cookies	291
get request info	293
get request infos	293
get request value	296
get response buffer	302
get response cookie domain	316
get response cookie expires	316
get response cookie http only	317
get response cookie path	318
get response cookie secure	318
get response cookie	313
get response cookies	314
get response header	321
get response headers	321
get response status	328
get root	394
get script timeout	336
get session array size	352
get session array	352
get session item	353
get session keys	353
get session stats	360
get session timeout	360
get session	351
get throw code	248
get throw message	248
get time remaining	332
get timestamp datetime	406
get upload content type	197
get upload encoding	197
get upload extension	198
get upload remote filename	198
get upload size	199
get utc delta	183
get version	332
getAt	474
getColumn	574
getData	575
getDefaults	532
getEmptyFields	522
getEnd	575
getMailMethod	493
getPersistentList	575

getPictureDescriptor	493
getPointerReferent	494
getRow	576
getSMTPAuthorization	494
getSMTPAuthPassword	495
getSMTPAuthUser	495
getSMTPHost	495
getStart	576
getStarts	533
getTimeout	576
getTitle	562
getUniqueID	522
getURLFactory	551
getVariablesIterator	522
global	248
globals has	208
globals	208
GOTO RECORD	341
GOTO SELECTED RECORD	342
gotoRow	577
handleError	551
handleErrorInline	552
hide session field	359
hideField	523
hideUniqueField	523
html encode	375
identical strings	372
image.a4d (script file)	261
import	249
in error	192
include into	250
include	250
insert into array	147
Insert string	372
insertAt	475
interpolate string	373
invalidAction	553
is a collection	166
is an iterator	218
is array	148
isAfterLast	577
isBeforeFirst	578
isFirst	578
isFuseboxRequest	553
isLast	578
join array	148
join paths	394
json encode	234

json to text.....	233
last not of	373
last of	374
last	475
last	579
left trim	374
len.....	476
library list	188
listToArray	476
load collection.....	157
local datetime to utc	183
local time to utc	184
local variables.....	415
lock globals	213
log message	193
longint to time.....	251
mac to html.....	375
mac to utf8	376
makeFuseboxLinks	533
makeLinks	534
makeURL.....	554
max of.....	257
maxRows	579
md5 sum.....	174
merge collections.....	159
method exists.....	252
min of.....	257
more items.....	216
MOVE DOCUMENT.....	395
move.....	579
multisort arrays.....	149
multisort named arrays.....	150
native to url path	395
new collection	154
new global collection.....	155
new json	220
new local collection	155
new	446
new	537
new	546
newFromArray.....	538
newFromArrays.....	580
newFromCachedSelection	581
newFromData	582
newFromFile.....	583
newFromRowSet	538
newFromSelection	539
newFromSelection	584

next item.....	216
next	540
next	589
nextID.....	496
nextId.....	595
nil pointer.....	252
object to collection.....	259
Open document.....	396
ORDER BY FORMULA.....	269
ORDER BY.....	268
ordinalOf.....	496
param text.....	376
parameter mode.....	333
parse json.....	236
parse.....	467
parseConfig.....	497
persistent.....	589
Position.....	378
postHandleError.....	554
prepend.....	476
previous.....	540
previous.....	589
purge.....	597
qualify.....	477
QUERY BY FORMULA.....	178
query params has.....	271
query params.....	271
QUERY SELECTION BY FORMULA.....	179
QUERY SELECTION.....	178
QUERY.....	177
QUERY/QUERY SELECTION.....	268
random between.....	257
read.....	595
RECEIVE PACKET.....	396
ReceiveCallback.....	71
redirect.....	253
regex callback replace.....	278
regex find all in array.....	279
regex find in array.....	280
regex match all.....	282
regex match.....	281
regex quote pattern.....	283
regex replace.....	284
regex split.....	287
request cookies.....	290
request info.....	293
request query.....	334
requested url.....	397

require	253
resize array.....	150
resolve path	397
RESOLVE POINTER.....	254
response buffer size	301
response cookies	313
response headers.....	321
rest.....	477
reverseArray	499
right trim.....	379
rowCount	590
save collection.....	157
save output.....	303
save upload to field.....	199
saveFormToSession	523
SELECTION/SELECTION RANGE TO ARRAY.....	150
selectionRangeToCollection	499
selectionToCollection	499
SEND PACKET.....	398
sendFuseaction.....	555
sendMail	501
session has.....	355
session id	357
session internal id	357
session local	358
session query	358
session to blob.....	350
session	350
sessionToBlob	503
set array.....	151
set cache control	325
set collection array	165
set collection	164
set content charset	328
set content type	327
set current script timeout	336
SET DOCUMENT POSITION	398
set error page.....	193
set expires date.....	326
set expires	326
set global array	212
set global	211
set http error page.....	194
set local	415
set log level.....	194
set output charset.....	304
set output encoding.....	305
set platform charset	334

SET QUERY DESTINATION	179
set response buffer	302
set response cookie domain	315
set response cookie expires	316
set response cookie http only	317
set response cookie path	317
set response cookie secure	318
set response cookie	314
set response header	322
set response status	328
set script timeout	335
set session array	355
set session timeout	359
set session	354
setAt	478
setColumnArray	590
setColumnData	591
setDefault	541
setDivId	547
setMailMethod	503
setRelateOne	591
setSeparator	547
setSMTPAuthorization	504
setSMTPHost	504
setTimeout	592
setTitle	563
setURLFactory	555
sleep	254
slice string	380
sort	478
sort	592
sourceRowCount	593
split path	398
split string	381
start json array	230
start json object	231
startArray	458
startObject	459
STRING LIST TO ARRAY	299
String	382
Substring	383
Table name	179
throw	255
time to longint	255
timedOut	593
timestamp date	405
timestamp day	407
timestamp difference	404

timestamp hour	408
timestamp millisecond	409
timestamp minute.....	408
timestamp month	407
timestamp second.....	409
timestamp string	405
timestamp time.....	406
timestamp year.....	407
timestamp	403
toJSON.....	461
trim	383
truncateText	504
type descriptor	416
undefined.....	416
unlock globals	213
unlockAndLoad.....	505
upload to blob.....	200
url decode path.....	384
url decode query	384
url decode	384
url encode path.....	385
url encode query	385
url encode	385
url to native path	399
utc to local datetime	184
utc to local time	184
utf8 to mac	386
validateTextFields	524
validEmailAddress.....	524
validPrice	506
valueCount	479
valueCountNoCase	479
valueList	479
variable name	417
warnInvalidField	525
week of year	185
write blob.....	307
write gif.....	264
write jpeg.....	265
write jpg	266
write json	233
write jsonp.....	234
write png	266
write raw.....	310
write to console	189
write	307
write	462
write	547

write	563
write	596
writeBold	525
writebr	309
writeln	309
writeln.....	310
writeln.....	462
yearMonthDay.....	507

APPENDIX B

ISO Language Codes

Afrikaans	af
Albanian	sq
Basque	eu
Bulgarian	bg
Belarusian	be
Catalan	ca
Chinese	zh
Chinese (Taiwan)	zh-tw
Chinese (PRC)	zh-cn
Chinese (Hong Kong)	zh-hk
Chinese (Singapore)	zh-sg
Croatian	hr
Czech	cs
Danish	da
Dutch (Standard)	nl
Dutch (Belgian)	nl-be
English	en
English (United States)	en-us
English (British)	en-gb
English (Australian)	en-au
English (Canadian)	en-ca
English (New Zealand)	en-nz
English (Ireland)	en-ie
English (South Africa)	en-za
English (Jamaica)	en-jm
English (Belize)	en-bz
English (Trinidad)	en-tt
Estonian	et
Faeroese	fo
Farsi	fa
Finnish	fi
French (Standard)	fr
French (Belgian)	fr-be
French (Canadian)	fr-ca

French (Swiss)	fr-ch
French (Luxembourg)	fr-lu
Gaelic	gd
German (Standard)	de
German (Swiss)	de-ch
German (Austrian)	de-at
German (Luxembourg)	de-lu
German (Liechtenstein)	de-li
Greek	el
Hindi	hi
Hungarian	hu
Icelandic	is
Indonesian	in
Italian (Standard)	it
Italian (Swiss)	it-ch
Japanese	ja
Korean	ko
Latvian	lv
Lithuanian	lt
Macedonian	mk
Malaysian	ms
Maltese	mt
Norwegian	no
Polish	pl
Portuguese (Brazilian)	pt-br
Portuguese (Standard)	pt
Rhaeto-Romanic	rm
Romanian	ro
Romanian (Moldavia)	ro-mo
Russian	ru
Russian (Moldavia)	ru-mo
Serbian	sr
Slovak	sk
Slovenian	sl
Sorbian	sb
Spanish	es
Spanish (Mexican)	es-mx
Spanish (Guatemala)	es-gt
Spanish (Costa Rica)	es-cr
Spanish (Panama)	es-pa

Spanish	es-do
Spanish (Venezuela)	es-ve
Spanish (Colombia)	es-co
Spanish (Peru)	es-pe
Spanish (Argentina)	es-ar
Spanish (Ecuador)	es-ec
Spanish (Chile)	es-cl
Spanish (Uruguay)	es-uy
Spanish (Paraguay)	es-py
Spanish (Bolivia)	es-bo
Spanish (El Salvador)	es-sv
Spanish (Honduras)	es-hn
Spanish (Nicaragua)	es-ni
Spanish (Puerto Rico)	es-pr
Sutu	sx
Swedish	sv
Swedish (Finland)	sv-fi
Thai	th
Tsonga	ts
Tswana	tn
Turkish	tr
Ukrainian	uk
Urdu	ur
Vietnamese	vi
Xhosa	xh
Yiddish	ji
Zulu	zu

APPENDIX C

Named Constants

Grouped by Function

Constant	Value
HTTP Status OK	200
HTTP Status No Content	204
HTTP Status Found	302
HTTP Status See Other	303
HTTP Status Not Modified	304
HTTP Status Bad Request	400
HTTP Status Unauthorized	401
HTTP Status Forbidden	403
HTTP Status Not Found	404
HTTP Status Request Timeout	408
HTTP Status Request Too Large	413
HTTP Status Server Error	500
A4D Log Off	0
A4D Log Critical Errors	1
A4D Log User Messages	2
A4D Log Execution Errors	4
Position From Start	1
Position From End	2
Position From Current	3
A4D Encoding None	0
A4D Encoding Quotes	1
A4D Encoding Tags	2
A4D Encoding Ampersand	4
A4D Encoding Extended	8
A4D Encoding HTML	8
A4D Encoding All	65535
A4D Error Type Unexpected	1

Constant	Value
A4D Error Type Syntax	2
A4D Error Type Database	3
A4D Error Type IO	4
A4D Error Type Critical	5
A4D Error Type Runtime	6
A4D Request Remote Addr	1
A4D Request Host Addr	2
A4D Request Host Port	3
A4D Request Secure	4
A4D Response Version	1
A4D Response Status	2
A4D Response Path	1
A4D Response Mod Date	2
A4D Response Mod Time	3
A4D Charset None	0
A4D Charset Mac	1
A4D Charset Win	2
A4D Charset ISO Latin1	3
A4D Charset Shift_JIS	4
A4D Charset GB2312	5
A4D Charset UTF8	6
A4D Not Executable	-1
A4D Parameter Mode Separate	0
A4D Parameter Mode Form	1
A4D Parameter Mode Query	2
A4D License Type Trial	0
A4D License Type Developer	1
A4D License Type Deployment	2
A4D License Type OEM	3
A4D License Type Special	4
A4D Regex Split No Empty	1
A4D Regex Split Capture Delims	2

Alphabetical

Constant	Value
A4D Charset GB2312	5
A4D Charset ISO Latin1	3
A4D Charset Mac	1
A4D Charset None	0
A4D Charset Shift_JIS	4
A4D Charset UTF8	6
A4D Charset Win	2
A4D Encoding All	65535
A4D Encoding Ampersand	4
A4D Encoding Extended	8
A4D Encoding HTML	8
A4D Encoding None	0
A4D Encoding Quotes	1
A4D Encoding Tags	2
A4D Error Type Critical	5
A4D Error Type Database	3
A4D Error Type IO	4
A4D Error Type Runtime	6
A4D Error Type Syntax	2
A4D Error Type Unexpected	1
A4D License Type Deployment	2
A4D License Type Developer	1
A4D License Type OEM	3
A4D License Type Special	4
A4D License Type Trial	0
A4D Log Critical Errors	1
A4D Log Execution Errors	4
A4D Log Off	0
A4D Log User Messages	2
A4D Not Executable	-1
A4D Parameter Mode Form	1
A4D Parameter Mode Query	2

Constant	Value
A4D Parameter Mode Separate	0
A4D Regex Split Capture Delims	2
A4D Regex Split No Empty	1
A4D Request Host Addr	2
A4D Request Host Port	3
A4D Request Remote Addr	1
A4D Request Secure	4
A4D Response Mod Date	2
A4D Response Mod Time	3
A4D Response Path	1
A4D Response Status	2
A4D Response Version	1
HTTP Status Bad Request	400
HTTP Status Forbidden	403
HTTP Status Found	302
HTTP Status No Content	204
HTTP Status Not Found	404
HTTP Status Not Modified	304
HTTP Status OK	200
HTTP Status Request Timeout	408
HTTP Status Request Too Large	413
HTTP Status See Other	303
HTTP Status Server Error	500
HTTP Status Unauthorized	401
Position From Current	3
Position From End	2
Position From Start	1