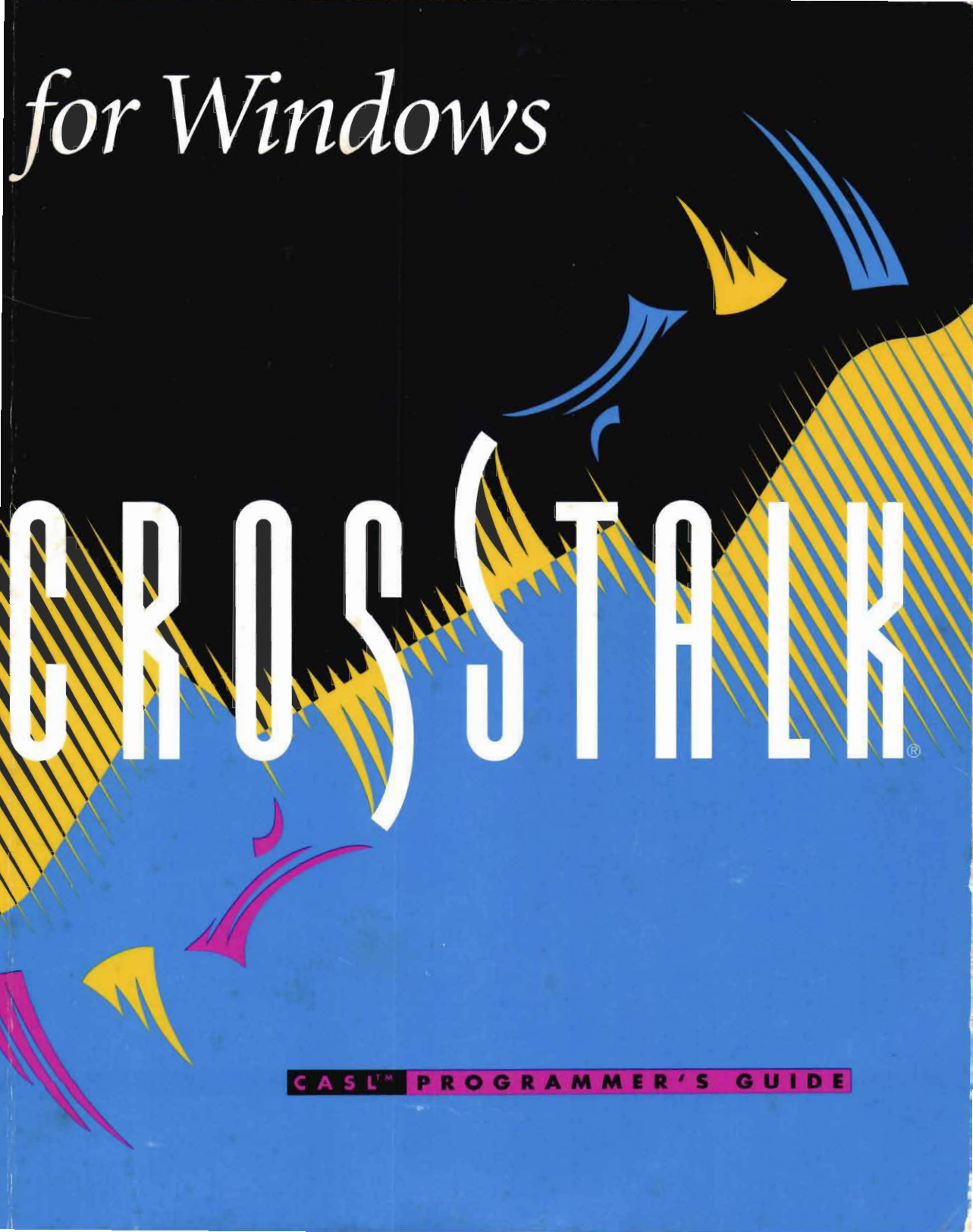
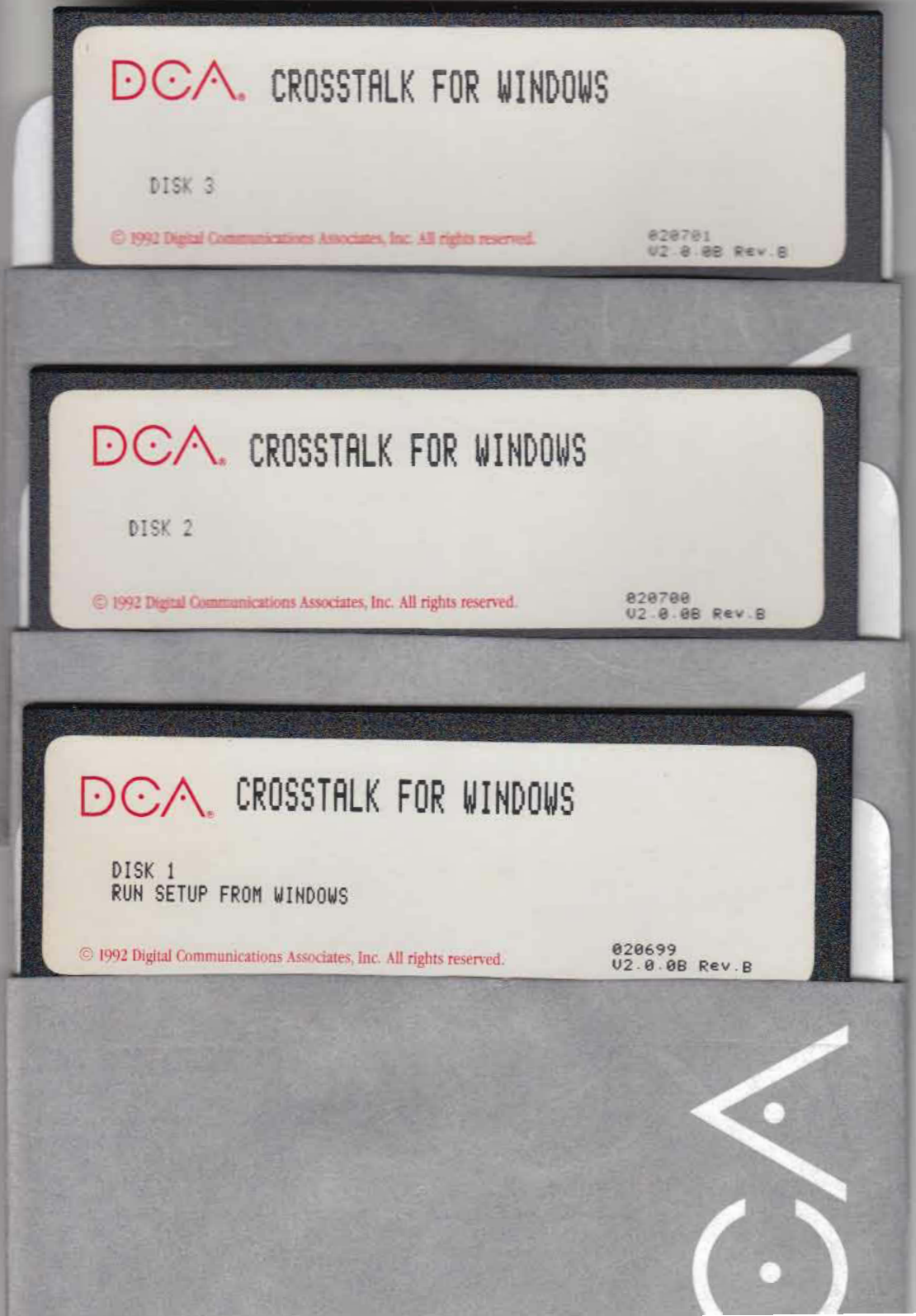
for Windows



CASL™ PROGRAMMER'S GUIDE


DCA CROSSTALK FOR WINDOWS
DISK 3
© 1992 Digital Communications Associates, Inc. All rights reserved.
020701
V2.0.0B Rev.B
DCA CROSSTALK FOR WINDOWS
DISK 2
© 1992 Digital Communications Associates, Inc. All rights reserved.
020700
V2.0.0B Rev.B
DCA CROSSTALK FOR WINDOWS
DISK 1
RUN SETUP FROM WINDOWS
© 1992 Digital Communications Associates, Inc. All rights reserved.
020699
V2.0.0B Rev.B

CASL
Programmer's Guide

Trademarks and registered names

Contents

Before You Begin ..... **xvii**

Intended audience xvii
About this guide xviii
Documentation conventions xix
Terminology xx
Common abbreviations xx
Need help? xxi
Using the DCA bulletin board xxi
Related publications xxii

1 Introducing CASL **1-1**

What is CASL? 1-2
How to use this guide 1-2
Why use scripts? 1-4
Scripts automate routine tasks 1-4
Scripts are easy to implement 1-4
Recording scripts with Learn 1-5
Recording keystrokes 1-5
Replaying your script 1-6
Writing scripts with CASL 1-6
Script types 1-6
Script structure 1-7
Comments 1-7
Declarations 1-7
Directives 1-8
Script elements 1-8
Statements 1-8
Variables 1-9
Constants 1-9
Expressions 1-9
Labels 1-9

Procedures and functions 1-9
Keywords 1-9
Designing a script 1-10
Developing a sample script 1-11
Logging on in a trouble-free environment 1-11
Describing the purpose of the script 1-12
Documenting the script's history 1-12
Displaying a message 1-12
Using string constants 1-13
Establishing communications with MCI Mail 1-13
Waiting for a prompt from the host 1-13
Sending the logon sequence 1-13
Using CASL predeclared variables 1-14
Using keywords 1-14
Ending the script 1-14
Using comments and blank lines 1-14
Verifying the MCI Mail connection 1-15
Declaring variables 1-16
Initializing variables 1-17
Performing a task while a condition is true 1-17
Using a relational expression to control the process 1-17
Waiting for a character string 1-18
Checking if a time-out occurred 1-18
Testing the outcome with a boolean expression 1-18
Branching to a different script location 1-19
Continuing the logon if the connection is established 1-19
Incrementing a counter using an arithmetic expression 1-20
Alerting the user if the connection failed 1-20
Disconnecting the session 1-20
Using indentation 1-21
Using braces with a statement group 1-21
Controlling the entire logon process 1-22
Performing a task while multiple conditions are true 1-24
Watching for one of several host responses 1-24
Sounding an alarm 1-26
Using the line-continuation sequence 1-27
Compiling and running your script 1-28
Compiling a script 1-29
From a communications session 1-29
From the Script Editor 1-29
Running a script 1-30
From a communications session 1-30
From the Script Editor 1-30
Where do you go from here? 1-31

2 Understanding the Basics of CASL 2-1

General rules for using CASL 2-2
Statements 2-2
Line continuation characters 2-2
Comments 2-3
Block comments 2-3
Line comments 2-3
Notational conventions used in this guide 2-4
Typeface 2-5
Angle brackets 2-5
Bold square brackets 2-6
Bold braces 2-6
Ellipsis 2-6
DOS and Macintosh differences 2-7
Terminology 2-7
Naming conventions 2-7
Script file name conventions 2-8
File path specifications 2-8
End-of-line delimiters 2-9
Wild cards 2-9
Identifiers 2-10
Data types 2-10
Integer 2-11
Real 2-11
String 2-11
Boolean 2-11
Byte 2-11
Word 2-11
Char 2-11
Array 2-11
Constants 2-12
Integer constants 2-12
Decimal integers 2-12
Hexadecimal integers 2-12
Octal integers 2-13
Binary integers 2-13
Kilo integers 2-13
Real constants 2-13
String constants 2-14
Embedded quotation marks 2-14
Unprintable characters 2-14
Special characters 2-16

Key names 2-16
String constants that continue on a new line 2-16
Boolean constants 2-16
Expressions 2-17
Order of evaluation 2-17
Arithmetic expressions 2-18
String expressions 2-21
String concatenation operation 2-21
Relational expressions 2-21
Boolean expressions 2-22
Type conversion 2-24
Converting an integer to a string 2-24
Converting a string to an integer 2-24
Converting an integer to a hexadecimal string 2-25
Converting an ASCII value to a character string 2-25
Compiler directives 2-26
Suppressing label information 2-26
Suppressing line number information 2-26
Trapping an error 2-26
Including an external file 2-27
Defining a script description 2-27
Reserved keywords 2-27

3 Declaring Variables, Arrays, Procedures, and Functions 3-1

Introduction 3-2
Variables 3-3
Predefined variables 3-3
System variables 3-3
Module variables 3-3
User-defined variables 3-4
Explicit declarations 3-4
Implicit declarations 3-5
Public and external variables 3-6
Initializers 3-6
Arrays 3-7
Single-dimension arrays 3-7
Arrays with multiple dimensions 3-7
Arrays with alternative bounds 3-8
Procedures 3-9
Procedure argument lists 3-9

Forward declarations for procedures 3-10
External procedures 3-11
Functions 3-12
Function argument lists 3-12
Forward declarations for functions 3-13
External functions 3-13
Scope rules 3-14
Local variables 3-14
Global variables 3-14
Default variable initialization values 3-14
Labels 3-15

4 Interfacing with the Host, Users, and Other Scripts 4-1

Interacting with the host 4-2
Waiting for a character string 4-2
Watching for one of several conditions to occur 4-3
Capturing data 4-4
Setting and testing time limits 4-5
Sending a reply to the host 4-6
Communicating with a user 4-6
Displaying information 4-6
Requesting information 4-7
Invoking other scripts 4-9
Chaining to another script 4-9
Calling another script 4-9
Passing arguments 4-9
Exchanging variables 4-10
Trapping and handling errors 4-11
Enabling error trapping 4-11
Testing if an error occurred 4-11
Checking the type of error 4-11
Checking the error number 4-12

5 Introducing the Programming Language 5-1

Functional purpose of CASL elements 5-2
Capture and upload control 5-2
Date and time operations 5-3
DDE interface 5-4
Device interaction 5-5

Error control 5-5
File input/output operations 5-5
Host interaction 5-7
Mathematical operations 5-8
Printer control 5-8
Program flow control 5-9
Script and session management 5-10
String operations 5-13
Type conversion operations 5-14
Window control 5-15
Miscellaneous elements 5-17

6 Using the Programming Language 6-1

Information provided for CASL elements 6-2
abs 6-3
activate 6-4
activatesession 6-5
active 6-6
activesession 6-7
add 6-8
alarm 6-9
alert 6-11
arg 6-13
asc 6-15
assume 6-16
backups 6-17
binary 6-18
bitstrip 6-19
blankex 6-20
breaklen 6-21
bye 6-22
call 6-23
capchars 6-24
capfile 6-25
capture 6-26
case/endcase 6-29
chain 6-31
chdir 6-32
chmod 6-33
choice 6-35
chr 6-36
cksum 6-37

class 6-38
clear 6-39
close 6-40
cls 6-41
cmode 6-42
compile 6-43
connected 6-44
connectreliable 6-45
copy 6-46
count 6-47
crc 6-48
curday 6-49
curdir 6-50
curdrive 6-51
curhour 6-52
curminute 6-53
curmonth 6-54
cursecond 6-55
curyear 6-56
cwait 6-57
date 6-59
definput 6-60
defoutput 6-61
dehex 6-62
delete (statement) 6-63
delete (function) 6-64
description 6-65
destore 6-66
detext 6-67
device 6-68
dialmodifier 6-70
dialogbox/enddialog 6-71
Dialog items 6-72
Dialog item options 6-74
dirfil 6-77
display 6-78
do 6-79
dosversion 6-81
downloaddir 6-82
drive 6-83
end 6-84
enhex 6-85
enstore 6-86
entext 6-87

environ 6-88
eof 6-89
eol 6-91
errclass 6-93
errno 6-94
error 6-95
exists 6-96
exit 6-97
extract 6-98
false 6-99
fileattr 6-100
filedate 6-102
filefind 6-103
filesize 6-104
filetime 6-105
fncheck 6-106
fnstrip 6-107
footer 6-109
for/next 6-110
freefile 6-113
freemem 6-114
freetrack 6-115
func/endfunc 6-116
genlabels 6-119
genlines 6-120
get 6-121
go 6-122
gosub/return 6-123
goto 6-125
grab 6-126
halt 6-127
header 6-128
hex 6-129
hide 6-130
hideallquickpads 6-131
hidequickpad 6-132
hms 6-133
if/then/else 6-135
include 6-138
inject 6-139
inkey 6-140
input 6-142
inscript 6-143
insert 6-144

instr 6-145
intval 6-146
jump 6-147
kermit 6-148
keys 6-150
label 6-151
left 6-152
length 6-153
linedelim 6-154
linetime 6-155
load 6-156
loadquickpad 6-157
loc 6-158
lowcase 6-159
lprint 6-160
lwait 6-162
match 6-164
max 6-165
maximize 6-166
message 6-167
mid 6-168
min 6-169
minimize 6-170
mkdir 6-171
mkint 6-172
mkstr 6-173
move 6-174
name 6-175
netid 6-176
new 6-177
nextchar 6-178
nextline (statement) 6-179
nextline (function) 6-181
null 6-183
number 6-184
octal 6-185
off 6-186
on 6-187
online 6-188
ontime 6-189
open 6-190
pack 6-192
pad 6-193
password 6-195

patience 6-196
perform 6-197
pop 6-198
press 6-199
print 6-201
printer 6-203
proc/endproc 6-204
protocol 6-207
put 6-209
quit 6-210
quote 6-211
read 6-212
read line 6-213
receive 6-214
redialcount 6-216
redialwait 6-217
rename 6-218
repeat/until 6-219
reply 6-221
request 6-222
restore 6-223
return 6-224
rewind 6-226
right 6-227
rmdir 6-228
run 6-229
save 6-230
script 6-231
scriptdesc 6-232
secno 6-233
seek 6-234
send 6-235
sendbreak 6-236
session 6-237
sessname 6-238
sessno 6-239
show 6-240
showallquickpads 6-241
showquickpad 6-242
size 6-243
slice 6-244
startup 6-245
str 6-246
strip 6-247

stroke 6-249
subst 6-250
systime 6-251
tabex 6-252
tabwidth 6-253
terminal 6-254
terminate 6-257
time 6-258
timeout 6-259
trace 6-260
track (statement) 6-261
track (function) 6-265
trap 6-267
true 6-268
unloadallquickpads 6-269
unloadquickpad 6-270
upcase 6-271
upload 6-272
userid 6-273
val 6-274
version 6-275
wait 6-276
watch/endwatch 6-279
weekday 6-282
while/wend 6-283
winchar 6-284
winsizex 6-285
winsizey 6-286
winstring 6-287
winversion 6-288
write 6-289
write line 6-291
xpos 6-292
ypos 6-293
zoom 6-294

7 Working with Terminal, Connection, and File Transfer Tools 7-1

The tool concept 7-2
Terminal tool 7-3
Connection tool 7-4
File transfer tool 7-5

8 Compatibility Issues 8-1

Introduction 8-2
Crosstalk for Windows 8-2
New elements 8-2
Changed elements 8-3
Removed elements 8-3
Crosstalk for Macintosh 8-4
Crosstalk Mark 4 8-4

A Windows Considerations A-1

Developing DDE scripts A-2
Topic name support A-2
Requesting information A-2
Executing Crosstalk commands A-3
Learning more about DDE A-5
DDE demonstration scripts A-6
Running the DDE scripts A-6
Information provided for DDE commands A-7
ddeack A-8
ddeadvise A-9
ddeadvisedatahandler A-10
ddeexecute A-12
ddeinitiate A-13
ddenak A-15
ddepoke A-16
dderequest A-17
ddestatus A-18
ddeterminate A-19
ddeunadvise A-20

B Macintosh Considerations B-1

Writing scripts for a Macintosh environment B-2

C Error Return Codes C-1

CASL error messages C-2
Compiler errors C-3
Input/output errors C-3

Mathematical and range errors C-4
State errors C-5
Critical errors C-6
Script execution errors C-7
Compatibility errors C-8
DOS gateway errors C-9
Call failure errors C-9
Missing information errors C-11
DDE errors C-12
Communications device errors C-13
Terminal errors C-13
File transfer errors C-14

D Product Support D-1

Requesting technical support D-2
Accessing DCA on-line services D-3
Updating or upgrading your software D-3

Index Index-1

Tables

1-1. Where to look for information 1-31

2-1. Placeholders in angle brackets 2-5
2-2. DOS and Macintosh terminology 2-7
2-3. ASCII control characters 2-15
2-4. CASL keywords 2-28

6-1. Alarm sounds 6-9
6-2. Integer values and their binary string lengths 6-18
6-3. Capture options 6-26
6-4. Bitmap values for the chmod statement 6-33
6-5. Class groupings 6-38
6-6. Options for the clear statement 6-39
6-7. Options for the cmode variable 6-42
6-8. Options for the cwait statement 6-57
6-9. Connection devices 6-68
6-10. Bitmap values for the fileattr function 6-100
6-11. Bitmap values for the fncheck function 6-106

6-12. Bitmap values for the fnstrip function 6-107
6-13. Bitmap values for the hms function 6-133
6-14. Keyboard keys and their corresponding numbers 6-140
6-15. Commands for the kermit statement 6-148
6-16. Parameters for the lwait statement 6-162
6-17. Integer ranges for the octal function 6-185
6-18. Mode options for the open statement 6-190
6-19. File transfer protocols 6-207
6-20. Terminal emulations 6-254
6-21. Conditions for the track statement 6-262
6-22. Conditions for the wait statement 6-277
6-23. Conditions for the watch statement 6-280

A-1. Valid requests for the system topic A-2
A-2. Valid requests for a session topic A-3
A-3. Valid commands for the system topic A-4
A-4. Valid commands for a session topic A-4
A-5. DDE demonstration script files A-6
A-6. DDE demonstration script control keys A-7

C-1. CASL error class values C-2
C-2. Input/output errors C-3
C-3. Mathematical and range errors C-4
C-4. State errors C-5
C-5. Critical errors C-6
C-6. Script execution errors C-7
C-7. Compatibility errors C-8
C-8. DOS gateway errors C-9
C-9. Call failure errors C-9
C-10. Missing information errors C-11
C-11. DDE errors C-12
C-12. Communications device errors—direct connection C-13
C-13. Terminal errors C-13
C-14. File transfer errors C-14

Before You Begin

The *CASL Programmer's Guide* is designed to assist you in creating and implementing scripts. It introduces the DCA® Crosstalk® Application Script Language (called CASL™) and explains how to use the language with your Crosstalk product.

The information provided is applicable to both the Macintosh® and Windows™ environments. Exceptions are noted in Chapter 8, "Compatibility Issues"; Appendix A, "Windows Considerations"; and Appendix B, "Macintosh Considerations."

Intended audience

This guide is written for users and programmers who want to write scripts using CASL. It provides conceptual information for the inexperienced programmer as well as detailed reference material for the sophisticated application developer.

Before reading this guide, you should have a knowledge of the following subjects:

- General concepts for the Crosstalk product you have installed. Refer to your product documentation for more information.
- One of the following operating environments:
  - Microsoft® Windows 3.1 with DOS 3.1 or newer.
  - Macintosh System 6.0.5 or newer. (Note that the Apple® Comm ToolBox is also required. Your Crosstalk product installs a copy if you do not already have one installed.)

About this guide

The *CASL Programmer's Guide* includes the following chapters:

Chapter 1, "Introducing CASL," contains information concerning why scripts are useful, how to create scripts by recording keystrokes, what makes up a script, how to develop a sample script, and how to compile and run a script.

Chapter 2, "Understanding the Basics of CASL," provides an understanding of the basic elements of CASL, such as identifiers, data types, constants, expressions, compiler directives, and CASL keywords. Notational conventions used to describe the CASL elements are explained in this chapter.

Chapter 3, "Declaring Variables, Arrays, Procedures, and Functions," covers how to declare elements in a script. Scope rules for variables and labels are also discussed.

Chapter 4, "Interfacing with the Host, Users, and Other Scripts," outlines some techniques you can use to interact with a host computer, communicate with a user, invoke other scripts, and trap and handle errors.

Chapter 5, "Introducing the Programming Language," provides a quick reference to the CASL elements grouped by their functional purpose.

Chapter 6, "Using the Programming Language," contains a detailed description, in alphabetical order, of each CASL element, with examples showing how each is used.

Chapter 7, "Working with Terminal, Connection, and File Transfer Tools," explains how to use variables to set up or modify Crosstalk's Terminal, Connection, and File Transfer tools.

Chapter 8, "Compatibility Issues," lists the language elements that are new, modified, or changed for Crosstalk for Windows. It also lists compatibility issues for Crosstalk for Macintosh and Crosstalk Mark 4.

Appendix A, "Windows Considerations," provides a detailed description of the Dynamic Data Exchange (referred to as DDE) commands supported by CASL.

Appendix B, "Macintosh Considerations," outlines considerations you should keep in mind when writing scripts for a Macintosh environment.

Appendix C, "Error Return Codes," contains tables of the error code values returned by Crosstalk.

Appendix D, "Product Support," explains the support provided by DCA.

This manual also includes an index.

Documentation conventions

The following documentation conventions are used in this manual:

KEY	This typeface represents a specific key on the keyboard. If you have remapped the function originally mapped to the key, substitute the new key or key combination for the original.
KEY1-KEY2	Keys displayed with a hyphen between them are called combination keystrokes. To enter combination keystrokes, press one key and hold it down while you press one or more other keys. Release all the keys at the same time.
`monospace text`	Monospace text is used to identify CASL elements. The elements can be CASL names, format descriptions, examples, and sample scripts.

Icons	Icons are used to show that text relates only to a particular subject. The following describes the icons that appear in this manual:
Win	Crosstalk for Windows
Mac	Crosstalk for Macintosh
Note:	This signifies important additional information.
Caution:	This symbol means that a failure to follow the recommended procedure could result in a loss of data or damage to equipment or related products.
■	This signifies the end of the text associated with a note, caution, or icon.

Terminology

The term pull-down, as used in this guide, refers to a pull-down menu in the Macintosh environment.

Common abbreviations

The following abbreviations are used in this guide.

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BBS	Bulletin Board Service
BPS	Bits per second
CASL	Crosstalk Application Script Language
CR	Carriage return
CRC	Cyclical redundancy check
CR/LF	Carriage-return/line-feed
DDE	Dynamic Data Exchange
DTE	Data Terminal Equipment
FCC	Federal Communications Commission
GUI	Graphical User Interface
KB	Kilobyte
KCP	Kermit Command Processor
NASI	NetWare® Asynchronous Services Interface

Need help?

If you have questions while using Crosstalk to edit, compile, or run a script, you can find the information you need in the on-line help. On-line help describes the purpose of a pull-down or dialog box; the available pushbuttons, list boxes, and edit boxes; and, where applicable, step-by-step instructions.

The information in the on-line documentation is both descriptive and instructive. That is, instead of merely stating the choices available, the on-line help guides you in making the correct choice.

For a detailed explanation of how to use the on-line help, refer to your Crosstalk user's guide.

Using the DCA bulletin board

You can stay informed about your Crosstalk product and communicate with other DCA users with the DCA Connection Bulletin Board System (BBS). You can learn about product announcements, news, and technical specifications; private and public e-mail; technical support, technical tips, and product histories; and a private/public user's file exchange for sending files to and from DCA engineers and other users.

Registration and access to the DCA Connection are free to all users by dialing this number with an asynchronous modem and using your Crosstalk or other asynchronous communications software:

(404) 740-8428

Set your communications software parameters as follows:

Data:	8
Parity:	N
Stop bits:	1
Speed:	1200, 2400, or 9600 (V.32)
Emulation:	ANSI (preferred) or TTY

Related publications

This manual does not provide a detailed explanation of the products, architectures, or standards developed by other companies or organizations. The following paragraphs indicate where to look for additional information.

For information on DOS, refer to the documentation provided by your DOS vendor.

For information on Microsoft Windows 3.1, refer to the documentation provided by Microsoft.

For information on Macintosh System 6 and System 7® and the Apple Comm ToolBox, refer to the documentation provided by Apple.

For information on Dynamic Data Exchange, refer to the *Microsoft Windows Software Development Kit, Guide to Programming 3.1.*

1 INTRODUCING CASL

What is CASL?	1-2
How to use this guide	1-2
Why use scripts?	1-4
Recording scripts with Learn	1-5
Writing scripts with CASL	1-6
Designing a script	1-10
Developing a sample script	1-11
Compiling and running your script	1-28
Where do you go from here?	1-31

What is CASL?

CASL is a scripting language that allows you to create custom scripts that can interface with other computers, users, and scripts. The scripts you develop can be simple or complex. For instance, you can create a simple script that waits for a prompt from the host computer and then replies with a user ID and password. Your more complex scripts can automate entire communications sessions or create custom menus that enable users to operate a host computer without learning its commands.

While CASL is designed to simplify the process of communicating with other computers, it is by no means limited to that function. CASL is a full-featured programming language that is capable of handling almost any task, including complex mathematical computations and the display of sophisticated dialog boxes. As you become familiar with its features, you will discover many other functions you can perform. The following section explains how to find what you need to create your CASL scripts.

How to use this guide

This guide is designed for easy use by beginners and experts alike. Depending on your programming expertise, you can start reading at different chapters.

If you are a beginner, start reading this chapter. It introduces CASL and explains why scripts are useful, how to record scripts with Learn, what makes up a script, and how to design and develop a script. It also describes how to compile and run a script.

If you already know about scripts and how to develop them, you can start reading Chapters 2 through 7. These chapters contain in-depth information about CASL and its comprehensive set of language elements. For your convenience, Chapter 5, "Introducing the Programming Language," contains a quick reference to the CASL elements. The elements are presented alphabetically by their functional purpose, and each has a brief one-line description.

If you have questions about CASL compatibility among Crosstalk products, refer to Chapter 8, "Compatibility Issues." This chapter covers compatibility issues for Crosstalk for Windows, Crosstalk for Macintosh, and Crosstalk Mark 4. Appendix A, "Windows Considerations," and Appendix B, "Macintosh Considerations," provide additional information specific to the Windows and Macintosh environments.

Once you start running your scripts, you may need to look up information about error messages. Appendix C, "Error Return Codes," contains a list of possible error return codes and what they mean.

The following chart shows at a glance where to find the information you need.

If you want to...	Read chapter(s)
Learn about scripts and how to record your keystrokes to create them	1
Develop a sample script	1
Compile and run a script	1
Review language reference material	2 to 7
Read about compatibility issues	8
Review Windows and Macintosh considerations	Appendix A and Appendix B
Look up error codes	Appendix C

Note: The term "host computer" is used throughout this guide. This term is used as a general reference to the remote system to which you are connected, regardless of the connection type (for example, modem, direct connection, NASI, or INT 14). A host can be another PC running a Crosstalk product, a system running a BBS program, or a large mainframe computer. ■

Why use scripts?

When you work in a data communications environment, you often have to perform the same functions over and over again to complete your daily activities. For instance, each time you start a communications session with the host computer, you have to enter your logon ID and password. In the following paragraphs, you will see how you can automate many routine tasks.

Scripts automate routine tasks

You can eliminate the manual repetition of routine tasks by using scripts to communicate with your host computer. You have to create and save a script to be able to use it; but once you have your script, you will find it invaluable in saving time and effort in the future. Furthermore, you will find that creating and implementing scripts are not difficult because CASL gives you an easy-to-use means of automating your daily activities within your computing network.

Scripts are easy to implement

Traditionally, developing applications and utilities that run in a communications environment required you to use a complex programming language and an Application Programming Interface (API) to access your host. You also had to understand the underlying data communications link. CASL removes these obstacles. When you write a CASL script, you do not have to concern yourself with the details of communications programming; CASL handles the communications interface. With CASL, you will discover how easy it is to automate many of the manual tasks you currently perform.

When you use CASL, you can create scripts that are simple, or you can develop complex scripts. You can create a script simply by recording the keystrokes you enter to log on to your host. In the following section, you will see how to use the Learn process to record scripts.

Recording scripts with Learn

You can record keystrokes to create scripts that perform routine activities. For example, you can create a script while you are entering your logon ID and password at your terminal.

Crosstalk's Learn facility captures the keystrokes you enter in a sequence of statements that are communicated to the host computer. You do not have to write any programming statements; the session connection and appropriate directives are incorporated for you by the script processor as part of the completed script. You can replay your recorded script just as it is, or you can use it as a base for developing a more complex script.

The sections that follow briefly describe the Learn process. For a more detailed description of how to use the Learn facility, refer to your Crosstalk user's guide and on-line help.

Recording keystrokes

When you are ready to enter your communications-session logon, or any other keystroke sequence, you can start recording a script. Follow these steps:

1. Start the Crosstalk application if it is not already active.
2. From a session window, choose Learn from the Script pull-down. Note that Learn changes to Stop Learn once you start the Learn process.
3. Type in the keystroke sequence you normally enter for the current communications session.
4. When you have completed the task, choose Stop Learn from the Script pull-down. At the prompt, specify the file name under which the script should be saved and also indicate whether the script should be set as the logon script.

The data you enter for your logon, or other communications function, is sent to the host as usual, but now you have a recorded script that you can replay to perform the same function.

Replaying your script

To replay your recorded script, follow these steps:

1 Start the Crosstalk application if it is not already active.

2 From a session window, choose Run from the Script pull-down.

3 Specify the script in the Run dialog box.

Note: If you set the script as a logon script, it is run automatically when the session connection is established. ■

Recording your keystrokes is a fast and efficient way to create scripts. However, you may want to write your own scripts using CASL. The following section provides guidelines to help you get started.

Writing scripts with CASL

Recording scripts allows you to automate many daily routines. However, you may want to create a script to handle special needs such as sending a file to the host computer or accessing information from a bulletin board service. To develop these scripts, use CASL.

CASL statements, functions, procedures, variables, and other language elements allow simple interaction with host-based systems. By following consistent guidelines for writing statements, you can make your script readable with the comprehensive set of keywords provided.

Script types

There are two main types of CASL scripts: on-line and off-line. On-line scripts work while Crosstalk is connected to a host and usually interact with the host to automate part of or an entire communications session. You can use on-line scripts to log on to the host, retrieve electronic mail, or create a custom menu interface for a host.

Off-line scripts do not interact with a host. For example, you can use an off-line script to display a list of host computers.

Note: A session is required to run either an on-line or an off-line script. ■

Script structure

CASL is flexible enough to accommodate most writing styles. If you have written computer programs before, you should be able to retain the same style you have used in the past.

In general, the contents of a script include such items as comments, declarations, and directives. Comments document a script; declarations define such items as variables, arrays, procedures, and functions; and directives specify an action to be taken.

Comments

Use comments to explain what will happen when a segment of code is executed or to block out part of a script that you do not want to execute. Comments are ignored by the script compiler and do not take up any space in a compiled script. Therefore, you can include as many comments as you feel necessary to document the purpose and flow of your script.

It is a good idea to start your script with a comment header that includes your name, the date of the script's creation, and some explanation of its objective. An example of this type of comment is as follows:

```
-- Script name:   LOGON.XWS
-- Date:          6/24/92
-- Author:        John Doe
```

In this example, the double dash is used to indicate a comment. Chapter 2, "Understanding the Basics of CASL," describes other notations you can use to designate a comment.

Declarations

Set up your declarations and assign values to them, if appropriate, immediately following the comment header. This will help you keep the declarations organized and easy to find, as shown in the following example:

```
-- Script name:   LOGON.XWS
-- Date:          6/24/92
-- Author:        John Doe

integer count, access_number

count = 1
access_number = NetID
```

Directives

The body of a script, which follows the declarations, is made up of directives, or statements. You can structure your script statements with one statement on a logical line, multiple statements on a logical line separated by colons (:), or a series of statements enclosed in braces ({ }). The following example shows one script statement on a logical line:

```
print "Hello!"
```

Chapter 2, "Understanding the Basics of CASL," provides examples of how to write statements using the alternate structures.

To make your script more readable and maintainable, you can indent statements that are part of a larger construct. Indentation, which is ignored by the compiler, is shown in the following example of a `for/next` construct:

```
-- This segment prints 1 through 10 vertically.
integer count
for count = 1 to 10
   print count
next
```

As shown in the preceding example, you can also use blank lines to improve program readability.

Script elements

Your scripts can consist of many different kinds of language elements. The sample script you develop in a later section contains examples of many of them. A brief description of the more commonly used CASL components follows.

Statements

Statements perform such functions as assignment of values, file input/output, file transfer, script flow control, host interaction, window control, and communications session management. CASL statements are described in detail in Chapter 6, "Using the Programming Language."

Variables

Variables are elements that can have different values from time to time. In your scripts, you can use variables that you create and variables that are predeclared by CASL. CASL's predeclared variables are described in Chapter 6, "Using the Programming Language."

Constants

Constants are elements that have a fixed value. Use the value directly in your script.

Expressions

Expressions include arithmetic expressions, string expressions, relational expressions, and boolean expressions.

Labels

Labels are named reference points in a script. A label can be the destination of a `goto` statement or it can mark the beginning of a subroutine. Guidelines for using the `label` statement in a script are presented in Chapter 6, "Using the Programming Language." Label scope rules are explained in Chapter 3, "Declaring Variables, Arrays, Procedures, and Functions."

Procedures and functions

Procedures and functions perform unique tasks. They differ in that functions return a value, and procedures do not. CASL provides built-in functions, which are predeclared. You can use these built-in elements as well as implement your own procedures and functions. See Chapter 6, "Using the Programming Language," for details.

Keywords

Keywords make your script more readable. CASL keywords are reserved for a particular use in your script; for example, statement names and words that bind arguments are all reserved keywords. You cannot use keywords as names for your variables, functions, procedures, or subroutines. Chapter 2, "Understanding the Basics of CASL," contains a table of the keywords reserved by CASL.

In the section "Developing a Sample Script" later in this chapter you will see how to use many of these elements in a script. Before you start creating a script, however, consider what you want your script to accomplish and how to structure the script to meet your programming objectives. The next section presents guidelines to help you design a script.

Designing a script

In the process of developing and implementing a more complex script, there is a typical development cycle. You will do the following, in the order shown:

- Design the script.
- Create and edit the script.
- Compile and locate errors.
- Fix the errors and compile again.
- Run the script; test it to be sure it works.
- Correct any problems and run the script again.

Before you actually begin to write a script, it is a good idea to map out what you want the script to accomplish. This step in the development cycle is especially important when you create scripts to use with communications programs. It is difficult to predict exactly what another computer will do during a communications session. Therefore, it is advisable to design your script to handle any type of situation that may occur.

Your script design can be as simple as a list of steps that outline the goals you want to accomplish. You can produce more detailed design plans by drawing flow charts. Listing goals and drawing flow charts are not always necessary, but they can often save you hours of work later.

When you have completed the initial framework, you are ready to write your script. Turn to the next section for guidelines on developing a script.

Developing a sample script

In an earlier section, you learned about recording keystrokes to create a script. This section explains how to develop a sample script using some of CASL's comprehensive set of language elements. To create a script, you need to use a text editor that produces plain ASCII text files, such as the one built into your Crosstalk product. Refer to your Crosstalk user's guide for information about the Script Editor.

The sample scripts that follow introduce you to the different forms of CASL statements, program design, and interaction with a host system that you can incorporate in a script. The samples are introduced in order of increasing complexity. An explanation of each sample script follows its presentation. The scripts are also provided on your Crosstalk distribution diskettes. Look for them in the main directory.

Logging on in a trouble-free environment

In this sample script, you send a logon sequence to MCI Mail. The example assumes that your script will run in a trouble-free environment, that is, it will not encounter errors or slow responses from the host.

```
/* This script shows how to display messages and send a user
ID and password to MCI Mail. */

-- Script name:  SAMPLE1.XWS
-- Created:      6/24/92 - Jane Smith

/* Display a message on the status line to tell the user
what is going on. */

message "MCI Mail auto-logon in progress"

/* Send a carriage return (CR) to get MCI's attention and
then send the logon user ID and password. */

reply                                  -- Send a CR
wait 2 seconds                         -- Wait for prompt
reply userid                           -- Send User ID
wait 2 seconds                         -- Wait for prompt
reply password                         -- Send password

message 'MCI auto-logon complete'      -- Tell the user

end                                    -- End the script
```

Describing the purpose of the script

At the beginning of the script, you find a comment describing the purpose of the script.

```
/* This script shows how to display messages and
send a user ID and password to MCI Mail. */
```

This type of comment is called a block comment because it is enclosed in the symbol pair /* and */. When you start your script with an explanation of its purpose, you make it easier for others to understand and use the script.

Documenting the script's history

As you can see, the sample script also contains a comment header that provides a history of the script's development, including the script name, the date it was created, and the author's name.

```
-- Script name:   SAMPLE.XWS
-- Created:       6/24/92 - Jane Smith
```

The header in this example shows the original date and author. For subsequent script modifications, the header might appear as follows:

```
-- Script name:   SAMPLE.XWS
-- Created:       6/24/91 - Jane Smith
-- Modified:      3/12/92 - Jane Smith
-- Modified:      7/16/92 - John Doe
```

Note that this comment is designated with a double dash. The double dash tells the script compiler that this is a line comment. Line comments do not require an end-of-comment symbol.

Displaying a message

In the first line of actual code, the sample script displays a message to tell the user what is occurring. To display this type of simple message, use the `message` statement.

```
message "MCI Mail auto-logon in progress"
```

Using string constants

As you can see in the foregoing `message` statement, the words that are displayed are enclosed in quotation marks. A character string enclosed in quotation marks is called a string constant. When you use CASL, you must enclose all string constants with quotation marks. You can use either double quotation marks, as shown in the preceding example, or single quotation marks, as shown in the script's second message.

```
message 'MCI auto-logon complete'
```

Be sure to use the same type of beginning and ending quotation marks.

Establishing communications with MCI Mail

To establish communications with MCI Mail, use the `reply` statement.

```
reply
```

When you use the `reply` statement without an argument, a carriage return is sent to the host application. This alerts the host to prompt for a user ID.

Waiting for a prompt from the host

After you send a carriage return to the host, you should wait for a brief period to allow the host to send a prompt.

```
wait 2 seconds
```

The `wait` statement causes the script to pause for `2 seconds` to allow the host to respond with the first prompt. Note that the amount of time to wait is dependent on your operating environment and the host.

Sending the logon sequence

Once you have set up the connection, you can send your user ID and password. To do this, use two `reply` statements—one to send the user ID and one to send the password. Be sure to wait for a brief period before sending the second `reply` statement to allow time for the host to send the password prompt.

```
reply userid
wait 2 seconds
reply password
```

Using CASL predeclared variables

CASL provides a rich set of predeclared variables, which include system variables and module variables. The sample script contains two of the predeclared system variables: `userid` and `password`.

`userid` and `password` are set up as system variables to make it easy for everyone to use CASL scripts and also to help maintain security. You can define these variables from the Crosstalk application by choosing Session from the Settings pull-down and then choosing the General icon. You can also modify these variables in a script. The sample script uses the predefined contents of the variables to send the user ID and password to MCI Mail.

```
reply userid
reply password
```

Using keywords

In the `wait` statement, you find the word `seconds`.

```
wait 2 seconds
```

This word is one of many CASL keywords that make your script more readable and flexible. Use the keywords only where specified in the various language elements.

Ending the script

There are several ways to end a script, depending on the reason for its termination. The most common way is to use the `end` statement, as shown in the sample script.

The `end` statement brings the script to an orderly conclusion. Other CASL statements, such as `halt`, `quit`, and `terminate`, cause related scripts, sessions, or the Crosstalk application to end also. These statements are discussed in detail in Chapter 6, "Using the Programming Language."

Using comments and blank lines

Throughout the sample script there are comments explaining what the programming code is to accomplish. Some of the comments are block comments, which are enclosed in the symbol pair `/*` and `*/`.

```
/* Display a message on the status line to tell the
user what is going on. */
```

Other comments are line comments.

```
-- Script name:  SAMPLE.XWS

reply                          -- Send a CR
```

As you can see, the line comments begin with a double dash (--). You can use both of these commenting methods in your script.

The sample script also shows how to use blank lines to make a script more readable. You can use blank lines almost anywhere in your script.

Verifying the MCI Mail connection

The preceding sample script assumed that MCI Mail responded to the initial carriage return within the expected time frame. This may not always be the case. This sample script shows how to verify that communications have, in fact, been established.

```
/* This script shows how to display messages and send a user
ID and password to MCI Mail.  it also verifies that the MCI
Mail connection is active. */

-- Script name:  SAMPLE2.XWS
-- Created:      6/24/92 - Jane Smith
-- Modified:     6/25/92 - Jane Smith  (Added code to
--                         check for the "port:" prompt.)

/* First, define the required variable. */

integer i

/* Display a message on the status line to tell the user
what is going on. */

message "MCI Mail auto-logon in progress"

/* Try to get MCI Mail's attention by sending a carriage
return (CR) until the "port:" prompt is received. */

i = 1                                  -- Initialize the
                                       -- variable to 1
while i <= 10                          -- Perform while i is
                                       -- less than or equal
                                       -- to 10
    reply                              -- Send a CR
    wait 2 seconds for "port:"         -- Wait for prompt
```

```
    if not timeout then                 -- If no timeout
    {
        goto LOGIN                      -- Branch to LOGIN to
                                        -- wait for prompts
    }
    i = i + 1                           -- Increment counter
wend

/* Could not get MCI Mail's attention.  Tell the user and
hang up. */

alert "System not responding - Logon canceled.", ok
bye                                     -- Disconnect
end                                     -- End

label LOGIN
wait for "name:"                        -- First prompt
reply userid                            -- Send user ID
wait for "password:"                    -- Next prompt
reply password                          -- Send password
message 'MCI auto-logon complete'       -- Tell the user

end                                     -- End the script
```

As in the first sample script, this sample starts with a description of its purpose and an outline of its history. (Note that the comment header has been updated to reflect a modification to the original script.) This script, however, adds logic to take into account that MCI Mail may not respond to the initial `reply` statement that sends a carriage return to the host.

First the script declares a variable that it will use as part of a conditional expression that determines how long to perform a task. As part of the task, it sends a carriage return to establish communications with MCI Mail and then waits for the expected character string from the application. If a time-out does not occur, the script branches to a different location to send the logon sequence to the application. If, however, communications cannot be established after 10 carriage returns are sent, the script alerts the user to the failure, disconnects the session, and ends.

Declaring variables

To declare a variable, specify a data-type identifier and a variable name. In the sample script, a variable named `i`, with a data type of `integer`, is declared.

```
integer i
```

This script uses only one variable. If your script contains multiple variables of the same data type, you can declare all of them on the same line.

```
integer i, tries
```

Note: If the variables have different data types, you must declare them on separate lines. ■

Initializing variables

The script compiler initializes an integer variable to a default value of zero. To initialize the variable to a different value, use the equal sign (=). In the sample script, the `i` variable is initialized to the value `1`.

```
i = 1
```

Performing a task while a condition is true

To execute statements repeatedly while a condition is true, use the `while/wend` construct. If the condition is initially false, the statements are not executed at all. This script uses the `while/wend` construct to control the process of connecting to MCI Mail.

```
while i <= 10
    reply
    wait 2 seconds for "port:"
    if not timeout then
    {
       goto LOGIN
    }
    i = i + 1
wend
```

The statements between the `while` and `wend` are continually executed until the condition `i <= 10` is no longer true. Then control passes to the statement following the `wend`.

Using a relational expression to control the process

Expressions that use relational operators such as "<=" are called relational expressions. When you use these operators, the result is always a boolean value (true or false). In this script, the relational expression `i <= 10` is used to determine how many times the `while/wend` construct is performed. As long as the condition is true, the statements within the construct are executed. When the condition is no longer true, the statement following the `wend` is executed.

Waiting for a character string

If you want your script to wait for one specific text string, use the CASL `wait` statement. This sample script waits for the character string `"port:"` to ensure that a connection with MCI Mail is established. To prevent the script from waiting forever, a duration time of `2 seconds` is specified.

```
wait 2 seconds for "port:"
```

You can determine if a time-out occurred before the character string arrived. The next section explains what to do.

Checking if a time-out occurred

Use the `if/then` construct and the `timeout` system variable to determine the outcome of the `wait` statement.

```
if not timeout then
{
   goto LOGIN
}
i = i + 1
```

The `timeout` system variable is either true or false indicating whether the last `wait` statement timed out. In this script, `timeout` is true if the `wait` statement exceeds the time specification of `2 seconds` before finding the `"port:"` text string.

When you use the `if/then` construct, the statement(s) following the `then` are executed only if the condition is true. In this script, the `goto LOGIN` statement is executed if a time-out does not occur; if a time-out occurs, the `i = i + 1` statement is executed.

Testing the outcome with a boolean expression

The condition you use in an `if ... then` statement is usually a boolean expression. Boolean expressions return either true or false. Your boolean expressions can be simple, as shown in this script:

```
if not timeout then
```

You can also use more complex expressions, involving multiple conditions with boolean operators, as shown in the following example:

```
if var1 >= 12 and var2 <= 5 then
```

In the sample script, if the boolean expression is true, the script transfers control to a logon routine, which is located in a different part of the script. The next section explains how to branch to a different script location.

Branching to a different script location

Sometimes it is preferable to handle a certain piece of coding logic in a separate part of a script. To branch to this location, you can use the `goto` statement.

```
if not timeout then
{
   goto LOGIN
}
```

To enable the script compiler to know where to branch, you must supply a `label` name in the `goto` statement. In the sample script, the label `LOGIN` is used to indicate the location where the next logical piece of code is located. The actual location is identified by the `label` statement.

```
label LOGIN
```

CASL provides another statement that allows you to branch to a label: `gosub ... return`. Chapter 6, "Using the Programming Language," describes this statement in detail.

Continuing the logon if the connection is established

If the script receives the `"port:"` prompt before a time-out occurs, it sends the logon sequence to the host, displays a message, and ends.

```
label LOGIN
wait for "name:"
reply userid
wait for "password:"
reply password
message 'MCI auto-logon complete'
end
```

If the "port:" prompt does not arrive in time, the script increments the while/wend conditional counter. Continue with the next section to learn how to use an arithmetic expression to increment a counter.

Incrementing a counter using an arithmetic expression

The number of times the while/wend construct is performed depends on the value in the variable i. To increment that value, you must use an arithmetic expression. Arithmetic expressions consist of numeric arguments and arithmetic operators. In the sample script, the addition operator, which is a plus sign (+), is used to add 1 to i.

```
i = i + 1
```

The counter continues to increment until the host sends the character string "port:" or until the counter's value no longer satisfies the condition for the while/wend construct (i <= 10) . If the host does not respond, the script alerts the user to the failure. Read the next section to learn about the alert statement.

Alerting the user if the connection failed

In general, the sample script uses the message statement to inform the user of current events. A message, which is displayed on its own without a dialog box, does not require any user intervention and is replaced by other messages.

To display information to which the user must respond, use the alert statement. The alert statement displays a message in a dialog box, which requires the user to choose a pushbutton to exit the dialog box. In the sample script, the alert statement provides an OK pushbutton for the user.

```
alert "System not responding - Logon canceled.", ok
```

The script pauses at the alert statement until the user chooses OK.

Disconnecting the session

If the connection with MCI Mail cannot be established, the script uses the bye statement to end the session. The bye statement immediately disconnects the current communications session and also disconnects the modem connection.

Using indentation

As you can see, some of the lines of code in the script are indented. For instance, the code within the `while/wend` loop is indented.

```
while i <= 10
    reply
    wait 2 seconds for "port:"
    if not timeout then
    {
        goto LOGIN
    }
    i = i + 1
wend
```

Indentation is not required, but it helps to make your script more readable. If indentation was not used in the sample script, it would be difficult to determine which lines of code applied to the `while/wend` construct.

Using braces with a statement group

You can use braces to enclose one or more statements that belong together. In the sample script, braces enclose the `goto` statement that follows the `if ... then` statement, indicating that the `goto` statement is part of the `if/then` construct.

```
if not timeout then
{
    goto LOGIN
}
```

Controlling the entire logon process

In the previous examples, the sample scripts did not verify the logon prompts sent by the host and therefore did not take corrective action if a prompt never appeared. In this script, you can see how to use the `watch/endwatch` construct, within a `while/wend` loop, to wait for any one of multiple character strings from the host and then take appropriate action based on the string that is received. The programming logic in this script gives you greater control over the sequence of events that may occur when communicating with your host computer.

```
/* This script shows how to display messages and send a user
ID and password to MCI Mail.  It also verifies that the MCI
Mail connection is active.  In addition, it uses the watch
statement to verify that the logon sequence is successfully
communicated to the host. */

-- Script name:  SAMPLE3.XWS
-- Created:      6/24/92 - Jane Smith
-- Modified:     6/25/92 - Jane Smith  (Added code to
--                         check for the "port:" prompt.)
-- Modified:     7/02/92 - John Jones  (Added code to
--                         check for specific logon
--                         prompts.)

/* First, define the required variables. */

integer i, tries

/* Display a message on the status line to tell the user
what is going on. */

message "MCI Mail auto-logon in progress"

/* Try to get MCI Mail's attention by sending a carriage
return until the "port:" prompt is received. */

i = 1                                   -- Initialize
                                        -- variable
while i <= 10                           -- Perform while i is
                                        -- less than or equal
                                        -- to 10
    reply                               -- Send CR
    wait 2 seconds for "port:"          -- Wait for prompt
    if not timeout then goto LOGIN      -- If no timeout,
                                        -- branch to LOGIN to
                                        -- check next prompts
    i = i + 1                           -- Increment counter
wend

/* Could not get MCI Mail's attention.  Tell the user and
hang up. */

alert "System not responding - Logon canceled.", ok
bye                                     -- Disconnect
end                                     -- End the script
```

```
label LOGIN                                  -- Branch-to location

/* Try to log on to MCI Mail for 50 seconds.  If not
successful, disconnect the session and exit. */

tries = 1                                    -- Initialize
                                             -- variable
while online and tries < 5                   -- Perform while both
                                             -- conditions are
                                             -- true
    watch 10 seconds for                     -- Wait for any one
                                             -- of the following
                                             -- host responses
        quiet 2 seconds  : reply
        "name:"          : wait 5 ticks : reply userid
        "password:"      : wait 5 ticks : reply password
        "sorry, inc"     : wait 5 ticks : bye : ...
          message "Unable to log on." : end
        "COM" : alarm 1  : message "MCI " + ...
          "Mail auto-logon complete." : end
        "call Customer Service" : ...
          alert "Connection refused.", ok : end
    endwatch
    tries = tries + 1                        -- Increment counter
wend

if tries < 5 then                            -- If not successful
{
    bye                                        -- Disconnect
    alert "Lost the connection.", ok           -- Tell the user
}
end                                          -- End
```

As in the second sample script, which verified the MCI Mail connection, this script contains the appropriate lead-in comments, attempts to establish communications with MCI Mail, waits for the `"port:"` prompt from the host, and branches to a different location to handle the balance of the logon process. At this point, however, this script uses a more comprehensive technique to ensure that it sends the correct logon responses to the host.

Based on two controlling conditions (the script is `online` and `tries` is less than 5), the script repeatedly watches for one of several host responses to arrive. If either of the two controlling conditions becomes invalid, the logon process terminates. Otherwise the script responds appropriately to whichever host prompt or message it receives.

Performing a task while multiple conditions are true

In the previous sample script, the `while/wend` construct contained one relational expression that determined how many times the while loop was repeated. This script uses two conditions to determine the duration of the loop: the result of the `online` function and the result of a relational expression.

```
while online and tries < 5
```

As long as both conditions are true, the statements in the `while/wend` construct are repeatedly executed. If either of the conditions becomes false, script execution continues with the statement following the `wend`.

The `online` function returns true as long as the script is on line to the host (that is, the modems are connected). The relational expression `tries < 5` returns true as long as `tries` is less than `5`. Since the variable `tries` is initialized to `1` before the while loop and then is incremented by `1` each time the loop is executed, the `while/wend` construct will be repeated a maximum of 4 times. It may be repeated fewer than 4 times, depending on what happens while the script is watching for one of several host responses.

Watching for one of several host responses

If you know that the host may send one of several different prompts, use the `watch/endwatch` construct with multiple conditions to watch for each possible prompt or message. The sample script watches 10 seconds for 6 potential conditions.

Write each watch condition as a separate entity. When one of the conditions occurs, the statements for that watch condition are executed and the `watch/endwatch` construct ends. If the 10-second time-out expires before a watch condition is satisfied, processing returns to the `while/wend` construct. If both of the while conditions are still true, the script executes the `watch/endwatch` construct again.

You need to write the actual `watch` statement only once for all of the watch conditions.

```
watch 10 seconds for
```

Each watch condition, along with its accompanying directives, is specified individually. These conditions are discussed in the paragraphs that follow. As you can see in this script, the watch conditions are followed by a colon (:). The colon is required.

A quiet connection

The first watch condition waits for the connection to be quiet for 2 consecutive seconds.

```
quiet 2 seconds : reply
```

If this condition is met, the script sends a carriage return to MCI Mail and processing returns to the `while/wend` construct. If the script is still `online` and `tries` is less than `5`, the `watch/endwatch` construct is executed again.

The "name:" prompt

The second watch condition looks for the character string `"name:"`

```
"name:" : wait 5 ticks : reply userid
```

If the script receives the `"name:"` prompt, it waits `5 ticks` (a tick is one tenth of a second) and then sends the contents of `userid` to MCI Mail. If the script is still `online` and `tries` is less than `5`, the `watch/endwatch` construct is executed again.

The "password:" prompt

If the host sends the `"password:"` prompt, the script executes the statements associated with the third watch condition.

```
"password:" : wait 5 ticks : reply password
```

After a brief wait of `5 ticks`, the script sends the contents of the system variable `password` to MCI Mail and then processing returns to the `while/wend` construct. The `watch/endwatch` construct is executed again if both of the while conditions remain true.

The "sorry, inc" message

The fourth watch condition looks for the character string "sorry, inc".

```
"sorry, inc" : wait 5 ticks : bye : ...
  message "Unable to log on." : end
```

If the script receives this message, it waits 5 ticks, disconnects the session, displays a message for the user, and ends. Processing does not return to the while/wend construct if this character string is received.

The "COM" message

If the host sends the "COM" message, the statements associated with the fifth watch condition are executed.

```
"COM" : alarm 1 : message "MCI " + ...
  "Mail auto-logon complete." : end
```

In this case, the script recognizes that the logon process has completed successfully. Therefore, it sounds an alarm to get the user's attention, displays an appropriate message, and ends.

The "call Customer Service" message

If the script receives the "call Customer Service" message, it executes the statements associate with the last watch condition.

```
"call Customer Service" : ...
  alert "Connection refused.", ok : end
```

The script displays an alert dialog box and waits for the user to choose the OK pushbutton; then it ends.

Sounding an alarm

To get the user's attention, you can use the alarm statement to make the terminal emit a sound. This script uses the alarm statement, with an argument of 1, to cause the terminal to play the "Close Encounters of the Third Kind" theme.

```
"COM" : alarm 1 : message "MCI " + ...
  "Mail auto-logon complete." : end
```

The `alarm` statement argument determines the type of sound that is heard. In this case, an argument of `1` specifies that the terminal should play the "Close Encounters of the Third Kind" theme. You can make the terminal sound other types of alarms, such as 3 beeps or a 4-note toot. Chapter 6, "Using the Programming Language," lists all of the possible alarm sounds.

Using the line-continuation sequence

To write a directive that continues on another line, use the line-continuation sequence (. . .) at the end of the line to be continued. You can see an example of this in the sample script.

```
"sorry, inc" : wait 5 ticks : bye : ...
   message "Unable to log on." : end
```

If you have a string constant that is too long to fit on one line, you can break the string into segments and use the line-continuation sequence to indicate the string continues on another line. You must enclose each string segment with quotation marks and use the string concatenation operator (+) to join the strings.

```
"COM" : alarm 1 : message "MCI " + ...
   "Mail auto-logon complete." : end
```

Compiling and running your script

Once you have created and saved a script, you should compile it to determine possible syntax errors. The script compiler converts your source script into a binary, machine-readable form and reports any errors that it detects. The compilation process takes only a small amount of time. When you have corrected all of the syntax errors, you can run the script.

Before you begin, however, it is important to understand how scripts are recognized by the script processor. Note the following:

- There are two types of script files: the source file, which you create and edit, and the executable file, which is created when you compile your script.
- To enable the script processor to differentiate between script source files and executable files, unique file-name formats are used.

  Your script source files are identified as follows:

  – By a .XWS file extension, if you are Windows user (LOGON.XWS)

  – By the file name alone, if you are a Macintosh user (LOGON)

  Your executable script files are identified as follows:

  – By a .XWC file extension, if you are a Windows user (LOGON.XWC)

  – By a bullet following the file name, if you are a Macintosh user (LOGON●)

Now you are ready to compile and run your script. The following sections explain how to proceed.

Note: To obtain detailed instructions for or assistance in compiling and running a script, use the on-line help provided with the Crosstalk software. ■

Compiling a script

You can compile a script from a communications session or from the Crosstalk Script Editor. The following sections explain how to proceed. Before you begin, be sure to save the script you have created.

From a communications session

To compile a script from a session window, follow these steps:

1. Start the Crosstalk application if it is not already active.
2. From a session window, choose Compile from the Script pull-down.
3. Specify the script in the Compile dialog box.
4. As the script compiles, make note of any compilation errors that may occur.
5. Correct the error(s).
6. Repeat steps 2 through 5 until your script compiles without errors.

From the Script Editor

To compile the script you are currently editing with the Crosstalk Script Editor, follow these steps:

1. Choose Compile from the Script pull-down or choose the CASL icon from the QuickBar. The Script Compiler message box, which displays the compiler's progress, is displayed.
2. Make note of compilation errors, if any should occur. (The compiler stops when a syntax error is encountered and allows you to exit to the Script Editor to correct the error. The error is highlighted to assist you in making corrections.)

   Note: To stop the compilation, choose Cancel from the message box. ■
3. Correct the error(s).
4. Repeat steps 1 through 3 until your script compiles without errors.

Note: The script compiler automatically compiles any script you run if the script has not already been compiled or if the most recent version of the source script is newer than the compiled version. However, we recommend that you compile your scripts before trying to run them to ensure that all syntax errors are corrected. ■

Running a script

You can run a script from a communications session or from the Crosstalk Script Editor. The following paragraphs explain each process.

From a communications session

To run a script from a session window, follow these steps:

1. Start the Crosstalk application if it is not already active.
2. From a session window, choose Run from the Script pull-down.
3. Specify the script in the Run dialog box.

Note: If you associate a script with a session when you define the session parameters, the script runs automatically when the session is started. ■

From the Script Editor

To run the script you are currently editing with the Crosstalk Script Editor, you must specify a session in which to run the script. To do this, choose a session from the Script pull-down. Note that only active sessions are displayed on the Script pull-down.

Once you start running your script, you do not have to actively participate other than to note run-time errors, if any should occur, or respond to prompts, if the script requires user input.

Note: You can use the Crosstalk trace facility while you are running a script. Tracing lets you track the lines of your script as they are executed. To start the trace facility, access a session window and choose Trace from the Script pull-down. When you activate tracing, the Trace option changes to Stop Trace. Choose Stop Trace to stop the trace facility.

You can also stop a running script from a session window by choosing Stop from the Script pull-down. ■

Where do you go from here?

In this chapter, you have been introduced to scripting and, in particular, to developing scripts using CASL. For some of you, the information provided is sufficient to satisfy the requirements of your job, and you know that you can create the scripts you need by using Learn to record your keystrokes.

For those of you who want to learn more about CASL, Table 1-1 can help you find the information you need.

Table 1-1. Where to look for information

To learn about...	Refer to...
Basic CASL concepts	Chapter 2
CASL's language elements	Chapters 5 and 6
Compatibility issues	Chapter 8
DDE scripts	Appendix A
Declarations for variables, arrays, procedures, or functions	Chapter 3
Error messages	Appendix C
Interfacing with a host, users, or other scripts	Chapter 4
Macintosh considerations	Appendix B
Product support	Appendix D
Sample scripts	Distribution diskettes
Terminal, connection, and file transfer tools	Chapter 7
Windows considerations	Appendix A

2 Understanding the Basics of CASL

General rules for using CASL	2-2
Identifiers	2-10
Data types	2-10
Constants	2-12
Expressions	2-17
Type conversion	2-24
Compiler directives	2-26
Reserved keywords	2-27

General rules for using CASL

CASL has general rules for using statements and comments in your script. This section outlines these rules and explains the notation used in this guide to describe the script language.

Statements

Statements specify an action to be taken. You can write the statements in any of the following ways:

- One statement to a logical line, as shown in the following example:

```
reply userid
```

- Multiple statements to a logical line with a colon (:) between each statement. This is shown in the following example:

```
wait for "Enter user ID:" : reply userid
wait for "Enter password:" : reply password
```

- A series of statements enclosed in braces ({ }), as shown in the following example:

```
if online then
{
   reply userid
   wait for "?"
   reply password
}
```

Line continuation characters

You can continue a statement on the next line by placing line continuation characters (...) at the end of the previous line. You can use the line continuation sequence anywhere in a script except inside quotation marks. The following example shows how to use the line continuation characters:

```
proc add_integers takes integer one_num, ...
   integer second_num
```

The line continuation sequence after the word `one_num` indicates that there is more information to follow.

Comments

Use comments to document your script. Comments are useful for maintaining, modifying, or debugging the script in the future.

You can add both block comments and line comments to a script. The following paragraphs explain each type.

Block comments

When you want to add a block of comments, enclose the comment text with the symbol pair /* and */ as shown in the following example:

```
/* This script logs on to the host.  First send the
host logon.  Then send the user ID and password.*/
```

You can use block comments anywhere in a script except in the middle of an identifier (such as a function or variable name) or inside a string constant. You can even nest comments in a block comment; the script processor sorts out the pairs correctly.

Be careful when using block comments, however, for if you fail to terminate the block comment correctly, the compiler will treat every statement in the rest of the script as part of the block comment.

Line comments

Use line comments when your comment text is brief. Line comments do not require a matching end-of-comment symbol.

There are two types of line comments—double hyphens (--) and the semicolon (;).

Note: We recommend that you use double hyphens for your line comments because the semicolon has special meaning for some of the CASL elements, such as the `print` statement. The semicolon comment indicator is supported only for backward compatibility. ■

Double hyphens

When you use the double-hyphen indicator, any characters that follow the hyphens, through the end of the line, are considered comment text. Since double hyphens are used only to designate a comment, you can use them anywhere (except, of course, in the middle of identifiers or string constants).

The following is an example of a double-hyphen comment:

```
-- Script name: HELLO.XWS
-- Date: 12-18-92
```

Semicolon

Use the semicolon indicator only in a location where you would normally place a CASL statement. The following are examples:

```
print "Hi," : ; This is a comment

reply userid
; Send your user ID to the host
```

Notational conventions used in this guide

Notational conventions are used to explain the syntax and semantics of the various procedures, functions, variables, and statements in the script language. The notation is only a typographical convention provided to help you understand how to use CASL and should not be used in your scripts.

The following notational conventions are used to illustrate the format of CASL language elements:

- Typeface
- Angle brackets
- Square brackets
- Braces
- Ellipsis

An explanation of the notation follows.

Typeface

Words or characters displayed in the following typeface are part of the script language:

```
online
```

Angle brackets

Words or characters in italics that are enclosed in angle brackets (< >) are placeholders for data you must fill in. The words or characters shown in the brackets often indicate the type of argument that is required. Table 2-1 explains some of the placeholders you may find in angle brackets.

Table 2-1. Placeholders in angle brackets

Word	Type	Explanation
<*char*>	Integer	The integer ASCII value of a character.
<*expression*>	Any	More than one type of expression can be used here. Read the text to determine which is suitable.
<*filename*>	String	A legal file specification. You can use full path names, as well as wild-card characters (where appropriate).
<*filenum*>	Integer	A file number. Range: 1–8. These expressions are usually optional and must be preceded by a pound sign (#) if they are specified.
<*time_expr*>	Integer	An amount of time. You can use any numeric expression followed by `ticks`, `seconds`, `minutes`, or `hours`. If you do not specify a keyword, `seconds` is assumed.

The following example illustrates the notational use of angle brackets:

```
delete <filename>
```

In this example, <*filename*> represents the name of a file.

Bold square brackets

Bold square brackets ([ ]) indicate that the argument is optional. The following example illustrates the notational use of bold square brackets:

```
close [# <filenum>]
```

In this example, the argument <filenum> is optional.

Bold braces

Words or characters in bold braces ({ }) represent multiple arguments from which to choose. The choices are separated by a vertical line, as shown in the following example:

```
genlines {on | off}
```

In this example, there are two choices, `on` and `off`. These are the only possible choices.

Ellipsis

An ellipsis (...) immediately after an item indicates that the previous item may be repeated. You can find an ellipsis used after items in angle brackets and after optional items in bold square brackets.

After an item in angle brackets

An ellipsis after an item in angle brackets indicates that you can repeat the previous item one or more times. The following example illustrates this notational use of the ellipsis:

```
<digit>...
```

In this example, you can have just one <digit>, or you may have multiple digits. You must have at least one digit.

After an optional item in bold square brackets

An ellipsis after an optional item in bold square brackets indicates that you can repeat the item zero or more times. The following example illustrates this notational use of the ellipsis:

```
[, <var>]...
```

In the preceding example, *var* is optional. If you choose to use *var* as an argument, the ellipsis indicates that you can have multiple variables as arguments.

Note: Parentheses, nonbold square brackets, and nonbold braces that appear in syntax descriptions and script language examples in this guide are part of the language and should be included in your script. ■

DOS and Macintosh differences

The information provided in this guide is applicable to both Macintosh and DOS environments. However, the two environments use different terminology and conventions. This section explains the differences.

Terminology

To simplify the presentation of information, this guide uses the DOS terminology in text. Whenever you see the DOS term shown in Table 2-2, it also refers to its Macintosh equivalent.

Table 2-2. DOS and Macintosh terminology

DOS	Macintosh
Drive	Volume
Directory	Folder
Subdirectory	Subfolder
File	File

Naming conventions

DOS drive names are limited to 1 character followed by a colon (for example, A:, B:, or C:). Directories and files are limited to 8 characters with an optional 3-character extension in the form `xxxxxxxx.xxx`.

Macintosh volume names can consist of up to 27 characters. Folder and file names can be up to 31 characters long.

Script file name conventions

To enable the Crosstalk script processor to differentiate between script source files and executable files, you must use distinctive file-name formats.

If you are creating a Windows script, use the following conventions:

- The .XWS file extension (LOGON.XWS) for source files
- The .XWC file extension (LOGON.XWC) for executable files

If you are creating a Macintosh script, use the following conventions:

- The file name alone (LOGON) for source files
- A bullet following the file name (LOGON●) for executable files

File path specifications

In a Windows script, use a backslash (\ ) to delimit drives, directories, and files. The following is an example:

```
"c:\xtalk\fil\somefile"
```

In a Macintosh script, use a colon (:) to delimit volumes, folders, and files. The following is an example:

```
"HD 80:Crosstalk:Download Folder:Some File"
```

This guide uses the DOS convention to represent both.

Absolute and relative file paths

An absolute file path is one that begins with the root directory while a relative file path starts with the current directory. The file paths shown in the preceding examples illustrate how to set up absolute path specifications for the DOS and Macintosh environments respectively.

To set up a relative file path for DOS, format the path as follows (assuming that `xtalk` is the current directory):

```
"fil\somefile"
```

To set up a relative file path for the Macintosh, format the path as follows (assuming that `Crosstalk` is the current folder):

```
":Download Folder:Some File"
```

Note that in a Macintosh environment, a colon must precede the first item specification; otherwise the first item is assumed to be the volume.

End-of-line delimiters

In a DOS environment, a carriage-return/line-feed (CR/LF) character is often used to indicate the end of a line. In a Macintosh environment, a carriage-return (CR) is used to designate the end of a line. This guide uses the DOS convention to represent both.

Wild cards

The DOS environment supports the use of wild cards (* or ?) to specify batch file operations. For example, if you want to send all of the files that have the .XWP extension to another computer, you can specify a wild-card file name as follows:

```
*.XWP
```

Although these wild cards are not a typical Macintosh convention, you can use them in a Macintosh CASL script to ensure the script is portable between the platforms.

Identifiers

Each variable, procedure, function, label, and other type of element used in a script must have a unique name, referred to as an identifier.

An identifier can be any length up to 128 characters. The first character must be alphabetic, or one of the following special characters: $, %, or _. The remaining characters can be alphabetic characters, special characters, or numbers; spaces cannot be used. Identifier names are not case-sensitive.

Unlike in some other programming languages (for example, BASIC), use of the percent (%) or dollar ($) symbol in a variable name does not force the variable to be a particular data type. CASL determines the data type of a variable from the keyword used in its explicit declaration or from the type of expression assigned to it in an implicit declaration. Refer to Chapter 3, "Declaring Variables, Arrays, Procedures, and Functions," for more information on variable declarations.

Note: Do not use the same identifier for different elements (for example, do not identify a variable with the same name assigned to a procedure). Duplicate identifiers are an error. ■

Data types

CASL supports the following data types:

- Integer
- Real
- String
- Boolean
- Byte
- Word
- Char
- Array

Note: For type-checking purposes, integer, byte, and word are all considered integers. ■

Integer

The integer data type represents positive and negative numbers. Internally, integers are stored as 32-bit signed integers, so values between -2,147,483,648 and 2,147,483,647 are possible.

Real

The real data type represents positive and negative floating point numbers. Internally, reals are stored as 4-byte IEEE floating point numbers, consisting of a sign bit, an 8-bit excess 127-bit binary exponent, and a 23-bit mantissa. The range of possible values is approximately 3.4E-38 to 3.4E+38.

String

The string data type represents variable length strings. A null string has zero length. The maximum length of any string is 32,767 characters.

A string variable has a particular length at any given time, but the length can change when a new value is assigned to the variable. The new length can be longer or shorter than the original length of the string.

Boolean

The boolean data type represents true or false values.

Byte

The byte data type consists of unsigned, non-fractional values of 0 (zero) to 255. It is often preferable to use bytes, rather than integers, in arrays because bytes require less memory than integers.

Word

The word data type consists of unsigned, non-fractional values from 0 (zero) to 65,535. As with the byte data type, you may find it preferable to set up your arrays using words, rather than integers.

Char

The char data type consists of a single-character string that can be assigned as strings or bytes.

Array

The array data type consists of multiple elements of a data type. You can have an array of integers, reals, strings, booleans, bytes, words, or chars.

Constants

A CASL constant can be one of the following four types:

- Integer
- Real
- String
- Boolean

Integer constants

Integer constants have one of the following formats:

`[-] <digit> ...`	Decimal integers
`[-] <digit> ... {h | H}`	Hexadecimal integers
`[-] <digit> ... {o | O | q | Q}`	Octal integers
`[-] <digit> ... {b | B}`	Binary integers
`[-] <digit> ... {k | K}`	Kilo integers

Decimal integers

Decimal integers use a base of 10, which means that `0` (zero) through `9` are valid digits. The following are examples of decimal integers:

```
1
-61
```

Hexadecimal integers

Integer constants that end with an `h` or `H` are hexadecimal constants. These constants use a base of 16; therefore, the digits of the constant can be `0` (zero) through `9` and also `a` through `f` (lower- or uppercase).

The first digit of a hexadecimal constant must always be numeric. If the leading digit is not numeric, you must supply a leading zero. The following are examples of hexadecimal constants:

```
0F0H
3f8h
```

Octal integers

Integer constants that end with the letter o, O, q, or Q are octal constants. These constants use a base of 8, which means that 0 (zero) through 7 are valid digits. The following are examples:

```
17o
17Q
```

Binary integers

Integer constants that end with a b or B are binary constants. Valid digits are 0 (zero) or 1 (one). Since the binary suffix b or B is also a valid hexadecimal digit, the script processor treats a b or B in an integer constant as a binary suffix only if the b or B is not followed by a legitimate hexadecimal digit or by the hexadecimal character h or H.

The following is an example of a binary constant:

```
1001001B
```

Kilo integers

Integer constants that end with a k or K are kilo integers. Valid digits for this type of integer constant are 0 (zero) through 9. When the script processor encounters a k or K following an integer constant, it multiplies the constant by 1,024. For example, 32K becomes 32,768.

The following are examples of kilo integers:

```
64K
128k
```

Real constants

Real constants specify a numeric value that may have a fractional component. For CASL to recognize a constant as a real constant, rather than as an integer constant, a decimal point (.) or the exponent indicator (e or E) must appear somewhere in it. A real constant must start with a digit (0 through 9) or a decimal point, optionally preceded by a minus sign.

Real constants have one of the following formats:

```
[-] [<digit>...] "." <digit>... [<exponent>]

[-] <digit>... <exponent>
```

The <*exponent*> has the following format:

```
{e | E} [+ | -] <digit>...
```

The following are examples of real constants:

```
0.2
-0.4e10
12.2e+10
20.3e-4
```

String constants

String constants consist of a string of characters enclosed in single quotation marks (') or double quotation marks ("). You must use the same type of beginning and ending quotation marks. A null string is represented as `''`, if you use single quotation marks, or `""`, if you use double quotation marks.

The following is an example of a string constant:

```
'This is a string'
```

In this example, the script processor recognizes that `This is a string` is a string constant because it is enclosed in single quotation marks.

Embedded quotation marks

If you have a quotation embedded in a string constant, use the other type of quotation marks to enclose the embedded quotation, as shown in the following example:

```
'She said, "Hello."'
```

In this example, the quotation `Hello` is enclosed in double quotation marks because it is embedded in a longer string, which is enclosed in single quotation marks.

Unprintable characters

To include an unprintable control character in a string constant, put a carat symbol before the control character (for example, `^G` for the control-G). To specify a numeric string, enclose the string in angle brackets (for example, `<007>` for the ASCII value 7). Table 2-3 lists the control characters and their corresponding ASCII values.

Table 2-3. ASCII control characters

ASCII	Control character	Description
0	^@	Null
1	^A	Start of header
2	^B	Start of text
3	^C	End of text
4	^D	End of transmission
5	^E	Enquiry
6	^F	Positive acknowledgment
7	^G	Bell
8	^H	Backspace
9	^I	Horizontal tab
10	^J	Line feed
11	^K	Vertical tab
12	^L	Form feed
13	^M	Carriage return
14	^N	Shift out
15	^O	Shift in
16	^P	Data link escape
17	^Q	Device control 1
18	^R	Device control 2
19	^S	Device control 3
20	^T	Device control 4
21	^U	Negative acknowledgement
22	^V	Synchronous idle
23	^W	End of transmission block
24	^X	Cancel
25	^Y	End of medium
26	^Z	Substitute
27	^[	Escape
28	^\	File separator
29	^]	Group separator
30	^^	Record separator
31	^_	Unit separator

Special characters

Some characters have special meanings. For example, the vertical bar (|) is interpreted as a carriage return; a single or double quotation mark is interpreted as a delimiter for a string constant; and a carat symbol is interpreted as notation for control characters.

If you want a special character to be recognized as part of the string rather than as a special character, use a backquote (`), which is also called a grave accent, before the special character. This is illustrated in the following examples:

```
reply "|"
reply "`|"
```

In the first example, the script processor interprets the " | " to mean a carriage return should be sent to the host. In the second example, the script processor recognizes that " ` | " means a vertical bar should be sent to the host.

If you want a backquote character to be recognized as part of the string, put two backquote characters in a row; the first one protects the second one.

Key names

If you need to specify a particular key on the keyboard, enclose the key name in angle brackets. Then enclose the entire string in quotation marks, as shown in the following example:

```
"<PF1>"
```

String constants that continue on a new line

If you have a string constant that is too long to fit on one line, break the string into segments, enclosing each segment with quotation marks, and use the string concatenation symbol (+) to join the segments. Do not use the line continuation sequence (...) or a carriage return inside the quotation marks. The following example illustrates how to continue a string constant on a new line:

```
message "You are running a new system " + ...
   "software version"
```

Boolean constants

A boolean constant is one of the following:

```
false
true
```

Expressions

CASL expressions include arithmetic, string, relational, and boolean expressions. There is a specific order of evaluation applied to these expressions based on precedence and the use of parentheses. A type conversion can be performed for some expressions. When a type conversion is performed, the original type of the expression is converted to a different type. Type conversion is explained later in this chapter.

Operators perform mathematical, logical, and string operations on expressions, or arguments. Most of the CASL operators have two arguments in the following format:

```
argument1 operator argument2
```

`argument1` and `argument2` must be expressions of the valid type for the operator involved. In general, you can use any expression containing a syntactically correct mixture of arguments and operators in a script wherever the result is allowed. For example, the following statements are functionally equivalent:

```
wait 9 seconds
wait 4 + 5 seconds
wait 3 * 3 seconds
wait 18 / 2 seconds
```

Order of evaluation

Expressions are normally evaluated based on the precedence of the operators; higher precedence operators are applied before lower precedence operators. You can control the order of evaluation of any expression by using parentheses. Subexpressions inside parentheses are evaluated before the main expression.

The general precedence of operators is as follows:

Highest precedence	Arithmetic and string operators.
Next highest precedence	Relational operators.
Lowest precedence	Boolean operators.

Arithmetic and string operators share the same precedence level because they cannot be mixed. Arithmetic and string expressions are completely evaluated before participating in relational expressions. Relational expressions are completely evaluated before participating in boolean expressions.

Within a particular type of expression, the precedence rules for that type are followed. The following sections explain the precedence rules for each expression.

Arithmetic expressions

You build arithmetic expressions using numeric arguments and arithmetic operators. Unary operators are evaluated from right to left, and binary operators of the same precedence are evaluated from left to right.

The standard arithmetic operators you can use are listed in groups of decreasing precedence. Each operator has a symbolic representation and a name.

The operators with the highest precedence are as follows:

```
~           BitNot
-           Negate
```

The operators with the second highest precedence are as follows:

```
rol         Rol
ror         Ror
shl         Shl
shr         Shr
```

The operators with the third highest precedence are as follows:

```
&           BitAnd
^           BitXor
/           Division
\           IntDivision
mod         Modulo
*           Multiplication
```

The operators with the lowest precedence are as follows:

```
+           Addition
|           BitOr
-           Subtraction
```

These operators, which are listed in alphabetical order, are explained in the paragraphs that follow.

`Addition` produces the numeric sum of its arguments. The following is an example:

```
2 + 2
```

`BitAnd`, `BitOr`, `BitXor`, and `BitNot` are bitwise operators. They are common operators in the assembler language. In the following diagrams, which show how these operators work, x and y are bit arguments and z is the result of the bitwise operation.

`BitAnd`

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

`BitOr`

x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

`BitXor`

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

`BitNot`

x	z
0	1
1	0

The following examples use `BitAnd`, `BitOr`, `BitXor`, and `BitNot`, in that order:

```
somevar = bitvar1 & bitvar2
somevar = somevar | bitvar3
somevar = somevar ^ bitvar3
somevar = ~ bitvar1
```

`Division` and `IntDivision` cause the mathematical division of the first argument by the second argument. For `Division`, the result is a real (floating point) value if either of the two quantities is a real; for `IntDivision`, only integers are allowed, and the result is an integer, possibly truncated. The following are examples:

```
x = 3.0 / 2.0                The result is 1.5
an_integer = 3 \ 2           The result is 1
```

`Modulo` returns the remainder after dividing its first argument by its second argument, as shown in the following example:

```
10 mod 4                     The result is 2
```

`Multiplication` is an algebraic operator that returns the product of two arguments. The following is an example:

```
2 * 2
```

`Negate` is also called "unary minus" in some programming languages. It multiplies a numeric value by minus one. The `Negate` operator is used in the following example:

```
neg_num = - pos_num
```

`Rol`, `Ror`, `Shl`, and `Shr` are bitwise operators that either rotate or shift the bits in an individual 8-bit, 16-bit, or 32-bit argument. These operators are common in the assembler language.

When you use these operators, the first argument has its value moved the number of positions specified in the second argument. In rotation, the bits that are moved off one end of the first argument are moved back onto the other end of the argument. In shifting, the bits that are moved off the end of the argument are discarded and replaced with zeros on the other end of the argument.

The `Rol` and `Shl` operators move bits to the left (toward the most significant bit) while the `Ror` and `Shr` operators move bits to the right (toward the least significant bit). The following are examples of these operators:

```
print 1 ror 8
print 1 shr 8
print 1 rol 8
print 1 shl 8
```

For the first example, '16,777,216' is printed. For the second example, '0' (zero) is printed. For the third and fourth examples, '256' is printed.

`Subtraction` reduces the first argument by the value in the second argument. Both arguments must be numeric. The following is an example:

```
4 - 2
```

String expressions

There is only one string operator—the string concatenation operator. However, CASL provides a comprehensive set of statements and functions that you can use to perform other string operations.

String concatenation operation

String concatenation joins two strings. The string concatenation operator is as follows:

```
+
```

When you use the string concatenation operator, two strings connected by a "+" are joined together to make one long string. This is shown in the following example:

```
"123" + "456"        is the string "123456"
```

For a complete list and description of the statements and functions that perform string operations, see Chapter 5, "Introducing the Programming Language," and Chapter 6, "Using the Programming Language."

Relational expressions

Relational expressions result in boolean values. The relational operators have no precedence.

You can use the following relational operators to compare numbers, strings, or booleans:

```
=                    Equal
>=                   GreaterOrEqual
>                    GreaterThan
<>                   Inequality
<=                   LessOrEqual
<                    LessThan
```

These operators are described in the paragraphs that follow.

`Equality` compares two expressions (either numeric or string) and returns `true` if the two items compared are exactly the same. Trailing spaces are significant in string comparisons. The following are examples of the `Equality` operation:

```
if a_variable = 2 then <statement>
```

Note: The equal sign is also used for variable assignment, as shown in the following example where the variable `a_variable` is assigned a value of `2`:

```
a_variable = 2■
```

`GreaterOrEqual`, `GreaterThan`, `LessOrEqual`, `LessThan`, and `Inequality` are also comparison operators. They apply to numeric quantities or strings. While the comparison of numeric quantities is straightforward, the comparison of strings is more complex.

In string comparisons, single characters are compared on the basis of their ASCII collating sequence; therefore, "Z" is less than "a." For longer strings, characters are compared position by position until a character is found that is different; then the characters that are different are compared on the basis of their ASCII collating sequence.

The following examples show the `LessThan`, `LessOrEqual`, `GreaterThan`, and `GreaterOrEqual` operators:

```
if some_var < 2 then <statement>
if string1 <= string2 then <statement>
while length(a_string) > 12
<statement> until rec_pointer => max_records
```

Boolean expressions

The boolean operators you can use are listed in the order of decreasing precedence.

The operator with the highest precedence is as follows:

```
not
```

The operator with the next highest precedence is as follows:

```
and
```

The operator with the lowest precedence is as follows:

```
or
```

The arguments to boolean operators can be boolean variables, relational expressions, or other boolean expressions.

`And`, `Or`, and `Not` produce a true or false result from their arguments, that is, they see their arguments only as true or false, not as quantities. The `And` operator returns true only if both arguments are true. The `Or` operator returns true if either or both of its arguments are true. The `Not` operator returns the opposite of its argument.

The following examples contain these operators:

```
if null(a_string) and x = 1 then <statement>
if counter > maximum or inkey then <statement>
if not eof(fl) and inkey <> 27 then <statement>
flip = not flip
```

If the value of the left argument of a logical operator is sufficient to determine the outcome of the expression, the right argument is not evaluated at all. This is the case when the left argument of the `And` operator is false, or when the left argument of the `Or` operator is true.

For instance, in the following example, the array reference `data[n]` will never attempt to index beyond the end of the array; if `n` were greater than 10, the expression `n <= 10` would be false, and the right argument would never be evaluated.

```
integer data[10]
if n <= 10 and data[n] >= 0 then <statement>
```

Type conversion

You may find it is necessary to convert values from one type to another. CASL provides the means to perform a variety of type conversions. This section explains how to convert an integer to a string, a string to an integer, an integer to a hexadecimal string, and an ASCII value to its corresponding character string.

Converting an integer to a string

To convert an integer to a string, use the `str` function. This function does not add leading or trailing spaces.

The following example illustrates how to use the `str` function:

```
reply str(share_to_buy)
```

In this example, `str` converts `share_to_buy` to a string, which is sent to the host with the `reply` statement.

Converting a string to an integer

To convert a string to an integer, use the `intval` function. This function ignores leading spaces and evaluates the string until a non-numeric character is found.

You can convert a string to a decimal or hexadecimal integer. If you need a hexadecimal integer, add an `H` to the end of the string. If your hexadecimal string does not begin with a numeric character, place a zero at the beginning of the string. If you need a kilo integer, add a `K` to the end of the string.

The following example illustrates how to use the `intval` function:

```
num = intval(user_input_string)
```

In this example, `intval` converts `user_input_string` to an integer and returns the result in `num`.

Converting an integer to a hexadecimal string

To convert an integer to a hexadecimal string, use the `hex` function. If the integer is below 65,536, the string is 4 characters long; otherwise, it is 8 characters long.

The following example shows how to use this function:

```
print hex(32767)
```

In this example, the `hex` function converts the integer 32,767 to a hexadecimal string and the result is displayed on the screen.

Converting an ASCII value to a character string

To convert an ASCII value to its corresponding 1-byte character string, use the `chr` function. The following is an example of how to use this function:

```
cr = chr(13)
```

In the preceding example, `chr` converts the ASCII value `13` to its corresponding carriage return character and returns the result in `cr`.

For more information on these and other CASL functions that perform type conversions, see Chapter 5, "Introducing the Programming Language," and Chapter 6, "Using the Programming Language."

Compiler directives

Compiler directives provide instructions for the script compiler. CASL compiler directives let you do the following:

- Suppress label information
- Suppress line number information
- Trap an error
- Include an external file
- Define a script description

Suppressing label information

By default, information about labels is included in the compiled version of your script. To suppress the label information, add the `genlabels off` compiler directive at the beginning of your source script. The default for this directive is `genlabels on`.

Note: If you use the `genlabels off` directive, you cannot use the `inscript` function or the `goto @<expression>` statement in your script. ■

Suppressing line number information

Information about line numbers is also included as part of a compiled script. To suppress this information, add the `genlines off` compiler directive at the beginning of your script. The default for this directive is `genlines on`.

Trapping an error

Use the `trap` compiler directive to enable and disable CASL's error trapping feature. Error trapping is disabled (`trap off`) by default. To enable error trapping, set `trap on` just prior to a statement that might generate an error. For additional information about trapping and handling errors, see Chapter 4, "Interfacing with the Host, Users, and Other Scripts."

Note: The `trap` compiler directive does not affect whether errors occur; it simply provides a way to effectively handle the errors if they do occur. ■

Including an external file

Use the `include` compiler directive when you want to include another file in the script being compiled. The file is included in the script following the `include` directive, as if the included file were part of the original file.

The `include` directive includes the file only once, no matter how many times you use the directive. The reason for this is that included files typically contain declarations, and including them more than once causes duplicate declaration errors.

Defining a script description

Use the `scriptdesc` compiler directive to define descriptive text for a script. When the script is added to the Script pull-down and to the Open dialog box, the `scriptdesc` text appears next to the associated script name.

For more detailed information about these compiler directives, see Chapter 6, "Using the Programming Language."

Reserved keywords

CASL reserves certain words called keywords. You may not use any of the keywords as identifier names. The reserved words are not case-sensitive.

Keywords include such elements as statements (for example, `capture` and `watch`), words that define time (for example, `seconds` and `ticks`), and words that bind statements, (for example, `for` and `next`).

Table 2-4, which begins on the following page, lists the CASL keywords.

Table 2-4. CASL keywords

abs	byte
accept	call
across	cancel
activate	capacity
activatesession	capchars
active	capfile
activesession	capture
add	case
alarm	cd
alert	chain
align	char
alluc	chdir
and	checkbox
answer	chmod
append	choice
arg	choices
arrow	chr
as	cksum
asc	class
assume	clear
at	close
attr	cls
aux	cmode
backups	color
binary	compile
bitstrip	connected
black	connectreliable
blankex	copy
blue	count
bol	crc
bool	ctext
boolean	curday
border	curdir
bow	curdrive
box	curhour
bright	urminute
breaklen	curmonth
brown	cursecond
browse	curyear
builtin	cwait
bye	cyan

continued

Table 2-4. CASL keywords (cont.)

date	editor
ddeack	edittext
ddeadvise	else
ddeadvisedatahandler	end
ddeexecute	endcase
ddeinitiate	enddialog
ddenak	endfunc
ddepoke	endproc
dderequest	endwatch
ddestatus	enhex
ddeterminate	enstore
ddeunadvise	entext
default	environ
definput	eof
defoutput	eoj
defpushbutton	eol
dehex	eop
delay	eow
delete	errclass
deletesubstring	errno
description	error
destore	exec
detext	exists
device	exit
devicevar	extern
dialmodifier	external
dialogbox	extract
dir	fail
direct	false
dirfil	field
diskspace	fileattr
display	filedate
do	filefind
dosversion	filesize
down	filetime
downloaddir	fill
draw	filter
drive	filtervar
drop	fkey
echo	flashing
edit	flood

continued

Table 2-4. CASL keywords (cont.)

fncheck	index
fnstrip	inject
focus	inkey
footer	input
for	inscript
form	insert
forward	instr
freefile	integer
freemem	intval
freetrack	inverse
from	is
func	isnt
function	istrackhit
genlabels	jump
genlines	keep
get	key
getnextline	keys
global	kermit
go	label
gosub	left
goto	leftjustify
grab	len
gray	length
green	library
group	lift
groupbox	line
alt	linedelim
header	linetime
height	listbox
help	load
hex	loadquickpad
hidden	loc
hide	locked
hideallquickpads	lowcase
hidequickpad	lprint
hms	ltext
hollow	lwait
hour	magenta
hours	match
if	max
include	maximize

continued

Table 2-4. CASL keywords (cont.)

maxlength	ontime
md	open
message	optional
mid	or
millisecond	output
milliseconds	over
min	pack
minimize	pad
minus	page
minute	paint
minutes	pan
mkdir	password
mkint	patience
mkstr	pause
mod	perform
modem	picture
move	plus
name	pop
netid	preserve
new	press
next	print
nextchar	printer
nextline	proc
noask	procedure
noblanks	prompt
nobye	protocol
nocase	protocolvar
none	public
nopause	pure
normal	pushbutton
not	put
null	quiet
number	quit
octal	quote
of	radiobutton
off	random
offset	rd
ok	read
on	real
online	receive
only	red

continued

Table 2-4. CASL keywords (cont.)

redialcount	show
redialwait	showallquickpads
release	showquickpad
remove	shr
rename	shut
repeat	size
replace	slice
reply	some
request	sort
reset	space
restore	start
resume	startup
return	statevar
returns	static
reverse	status
rewind	step
right	str
rmdir	string
rol	strip
ror	stripclass
routine	stripwild
rtext	stroke
run	style
save	subst
script	subtitle
scriptdesc	swap
scroll	systemvar
secno	systime
second	tabex
seconds	tabstop
seek	tabwidth
send	takes
sendbreak	terminal
session	terminalvar
sessionvar	terminate
sessname	then
sessno	tick
setup	ticks
setvar	time
shl	timeout

continued

Table 2-4. CASL keywords (cont.)

times	watch
title	weekday
to	wend
toggle	while
trace	white
track	width
trackhit	winchar
trap	window
true	winsizex
type	winsizey
unloadallquickpads	winstring
unloadquickpad	winversion
until	word
up	write
upcase	xpos
upload	xsep
userid	yellow
val	yourself
version	ypos
view	ysep
viewport	zone
wait	zoom

3 Declaring Variables, Arrays, Procedures, and Functions

Introduction	3-2
Variables	3-3
Arrays	3-7
Procedures	3-9
Functions	3-12
Scope rules	3-14

Introduction

In Chapter 2, "Understanding the Basics of CASL," you were introduced to the basic components of CASL. As you develop your scripts, you will find it necessary to declare many of these elements, just as you declare them in other programming languages.

In a CASL script, you use declarations to define your variables, arrays, procedures, and functions. Declarations make your script more readable and maintainable; in some instances, they are mandatory. The information contained in this chapter will help you understand and use declarations.

Variables

A variable is a language element whose value can change during the course of running a script. You use variables as storage areas where you can keep the results of a computation, data arriving from the host, and other data such as a user name or password.

With CASL, you can use two types of variables: predefined variables, which you can reference in your script; and user-defined variables, which you define in your script.

Predefined variables

There are two types of predefined variables: system variables and module variables.

System variables

System variables contain user-profile (or configuration) information or session information.

The variables that contain user-profile information are stored in the XTALK.INI file on a PC or in "Crosstalk Preferences" in the Preferences folder, which is in the System folder, on a Macintosh. The information in these variables is global, that is, it pertains to all sessions.

The variables that contain session information are stored in a session profile. Each session entry contains session parameters such as the terminal emulation type, user ID, and password.

Module variables

Module variables contain tool-specific information and are stored in a session profile. For example, if a session uses the `dcamodem` connection device type, the entry contains settings for Port, Speed, DataBits, and so on. To reference these variables, use the `assume` statement as follows:

```
assume device "DCAMODEM"
```

User-defined variables

User-defined variables are those you define in your script. These variables can be local to one script or shared across multiple scripts.

You must declare your variables before you use them. With CASL, you can declare them explicitly or, in some cases, implicitly.

Explicit declarations

Explicitly declare your variables to make your script more readable and maintainable.

Explicit declarations consist of a data-type identifier and a variable name. You can use any variable name you like as long as it is not the same as that of another language element in your script. It is often helpful to assign a name that reflects the variable's purpose; for example, the name `file_name` is more descriptive than the name `xyz`.

Your variable names can contain any combination of alphanumeric characters as well as some symbols. The first character must be alphabetic, or one of these special characters: $, %, or _. Variable names can consist of up to 32,767 characters.

The following illustrates the general form of an explicit declaration:

```
<data_type> <name> [, <name>]...
```

Single-variable declarations

You can declare variables one to a line. The following is an example of single declaration:

```
integer counter
```

In this example, `counter` is declared as an integer variable.

Multiple-variable declarations

You can also declare more than one variable on a logical line, but the variables must be of the same type. Multiple declaration is shown in the following example:

```
integer row, col
```

In this example, both `row` and `col` are declared as integer variables.

The following are examples of explicit declarations for other data types:

```
boolean failed
real percentage
string file_name, extension
```

Implicit declarations

You can implicitly declare a variable if the first time it is used it is possible to infer its type from the context. However, use implicit declarations sparingly, for your script is less readable and maintainable when variables are not declared explicitly.

The most common case of implicit declaration is where the variable is assigned a value. In this case, the type of the variable is implicitly declared to match the type of the expression assigned to it. In the example that follows, `user_name` is implicitly declared as a string variable because the string `"John"` is assigned to it. Note that `"John"` is enclosed in quotation marks; you must use quotation marks to enclose a data string assigned to a string variable.

```
user_name = "John"
```

The same concept applies for all other cases where the variable type can be inferred. For instance, the following example implicitly declares `count` to be an integer variable because the initial value is an integer.

```
for count = 1 to 10
     -----
     -----
next
```

Public and external variables

If you want to share a variable among multiple scripts, declare the variable as `public` in the main script (parent script) and as `external` in the other scripts (child scripts). The data type of the variables must match; and if the variable is an array, the declared array size must match. As with any other explicit declaration, you can declare multiple public or external variables of the same type on one logical line, separating the variable names with commas.

The following are examples of public and external variables:

```
public integer user_name          (parent script declaration)
external integer user_name        (child script declaration)
```

For additional information about public and external variables, see Chapter 4, "Interfacing with the Host, Users, and Other Scripts."

Initializers

Variables you declare explicitly are automatically initialized by the compiler: strings are initialized to nulls; reals and integers are initialized to zero. To initialize these variables to a different value, use the assignment operator (=).

The following are examples of variable initialization:

```
a_var = 10
amount = "Quantity"
```

In the first example, the integer variable `a_var` is initialized to `10`. In the second example, the string variable `amount` is initialized to `Quantity`.

Arrays

Arrays require an explicit declaration; it is not possible to implicitly declare an array.

An array declaration is similar to other declarations, but you must also declare the dimensions. Enclose the dimensions of the array in square brackets.

Note: The elements in CASL arrays are numbered starting from zero; therefore, there are actually $n + 1$ elements in an array of size n. ■

Single-dimension arrays

Some arrays have only one dimension. For example, you declare a single-dimension array of 30 integers as follows:

```
integer epsilon[29]
```

In this example, the size of the array `epsilon` is `29`, but there are actually 30 elements in the array because the first element is element 0 (zero).

Arrays with multiple dimensions

Arrays can also be multidimensional. You declare multiple dimensions by providing multiple dimension sizes, separated by commas. For example, you declare a 10-by-20 string matrix in the following way:

```
string matrix[9, 19]
```

Arrays with alternative bounds

You can use alternative bounds declarations when you need to use bounds other than the default. The following examples show how to declare arrays with alternative bounds:

```
integer vector[0:99]
integer profile[3:6]
integer samples[-10:10]
```

The first example, an array of 100 elements, is equivalent to `integer vector[99]` because `0` is the default lower bound. In the second example, the array `profile`, an array of 4 elements, is indexed from 3 to 6. The array `samples`, an array of 21 elements, is indexed from -10 to 10 in the third example.

When you declare multiple dimensions, you can use alternative bounds declarations for each dimension individually. For example, declare a matrix whose first dimension is indexed from 10 to 30 and whose second dimension contains 100 integers in the following way:

```
integer data[10:30, 99]
```

Procedures

A procedure definition is a declaration because it only defines the statements that make up the procedure. The statements themselves are not executed until the procedure is called.

You must declare a procedure before you use it. A procedure cannot be inside a function or another procedure.

Procedures are useful for replacing groups of statements that are frequently used. For example, a script that repeatedly performs a complicated sequence of steps can use one common procedure to perform the task. The statement(s) that call the procedure simply pass the appropriate information to the procedure, and it performs the task. If you need to return a result, consider using a function instead of a procedure.

The following example illustrates the syntax of a procedure definition:

```
proc <name> [takes <arglist>]
     -----
     -----
endproc
```

Procedure argument lists

As shown in the preceding syntax illustration, a procedure can have an argument list. The `<arglist>` is optional, and is used only if the procedure takes arguments. If arguments are included, you must use the same number and type of arguments in both the procedure and the statement that calls the procedure. The arguments are assumed to be strings unless otherwise specified.

The syntax of `<arglist>` is as follows:

```
[<type>] <argument> [, [<type>] <argument>]...
```

The following is an example of a procedure definition:

```
/*
This procedure sends the user ID and password to the
host.
*/

proc logon takes username, passwrd
   reply username
   wait 2 seconds
   reply passwrd
endproc
```

In this example, the statements enclosed in the `/*` and `*/` symbols are comments describing the procedure's purpose. The procedure, which is named `logon`, expects two string arguments—`username` and `passwrd`; and it sends the arguments to the host. When the procedure ends (`endproc`), control is passed to the statement immediately following the one that called the procedure.

You call this procedure as follows:

```
logon userid, password
```

The arguments `userid` and `password` are passed to the procedure `logon`.

Forward declarations for procedures

You can use forward declarations to declare procedures whose definitions occur later in the script. The syntax of a forward procedure declaration is the same as the first line of a procedure definition, with the addition of the `forward` keyword.

Forward declarations are useful if you want to place your procedures near the end of your script. A procedure must be declared before you can call it; the forward declaration provides the means to declare a procedure and later define what the procedure is to perform.

The following syntax is used for a forward declaration:

```
proc <name> [takes <arglist>] forward
```

When the procedure definition is encountered, each of its arguments (if provided) must match the data type of the corresponding argument in the forward declaration.

The following example illustrates how to set up the `logon` procedure using a forward declaration:

```
proc logon takes ...                    -- The forward declaration
   username, passwrd forward

logon userid, password                  -- The procedure call

proc logon takes username, passwrd      -- The procedure
   reply username
   wait 2 seconds
   reply passwrd
endproc
```

You can also use the `perform` statement to call a procedure before it is declared. This is shown in the following example:

```
perform logon userid, password
```

External procedures

Procedures can be an integral part of a script, or they can be in separate files. The latter allows you to keep a library of procedures you often use; you don't have to duplicate the procedure for each script you create.

To include an external procedure in a script, use the `include` compiler directive. For example, suppose the `logon` procedure, which was described previously, is an external procedure that is stored in a file called `myprocs.xws`. To include it in your script, add the following line at the beginning of the script:

```
include "myprocs"
```

For more information about the `proc/endproc` procedure construct, the `perform` statement, and the `include` compiler directive, see Chapter 6, "Using the Programming Language."

Functions

A function is similar to a procedure, but it returns a value. You must declare the type of the return value within the function definition and specify a return value before returning.

You must declare a function before you can use it. A function cannot be inside a procedure or another function.

The syntax of a function definition is as follows:

```
func <name> [(<arglist>)] returns <type>
     -----
     -----
endfunc
```

Function argument lists

As for a procedure, the <arglist> is optional. The syntax of the <arglist> is the same as for procedure arguments.

The following example illustrates a function with an <arglist>:

```
func calc(integer x, integer y) returns integer
     if x < y then return x else return y
endfunc
```

In this example, the integers x and y are the function arguments. The values of x and y are passed to the function when it is called. The function returns one or the other value depending on the outcome of the if then else comparison. If x is less than y, x is the return value; if x is not less than y, the value of y is returned.

You call this function as follows:

```
integer return_value

return_value = calc(3, 8)
```

The integer values of 3 and 8 are passed to the function calc where they are used as the values x and y in the function. The function returns the result of its calculations in the variable return_value.

Forward declarations for functions

You can use forward declarations to declare functions whose definition occurs later in the script. The syntax of a forward function declaration is the same as the first line of a function definition, with the addition of the `forward` keyword.

Forward declarations are useful if you want to place your functions near the end of your script. A function must be declared before you can call it; the forward declaration provides the means to declare a function and later define what the function is to perform.

The following syntax is used for a forward declaration:

```
func <name> [(<arglist>)] returns <type> ...
   forward
```

When the function definition is encountered, each of its arguments (if provided) must match the data type of the corresponding argument in the forward declaration.

The following example illustrates how to set up the `calc` function using a forward declaration:

```
integer return_value                    -- The integer declaration
func calc(integer x, integer y) ...             -- The
   returns integer forward                      -- forward
                                                -- declaration

return_value = calc(3,8)                    -- The function call

func calc(integer x, integer y) ...             -- The function
     returns integer
     if x < y then return x else return y
endfunc
```

External functions

As with procedures, functions can be in separate files. To include an external function in a script, use the `include` compiler directive. For example, if the `calc` function is external to the script and is stored in a file called `myprocs.xws`, add the following line at the beginning of the script to include it in the script:

```
include "myprocs"
```

For more information about the `func/endfunc` function construct and the `include` compiler directive, see Chapter 6, "Using the Programming Language."

Scope rules

You can reference a variable from the line on which it is declared until the end of its scope. This is true for both implicit and explicit declarations.

Local variables

The variables you declare inside procedures and functions are local variables. The scope of local variables terminates when the function or procedure that defines them ends. You can refer to and modify these variables only while the procedure or function is executing. Their values are lost when the procedure or function returns control.

Global variables

The variables you declare outside procedures and functions are global variables. The scope of global variables terminates when the script ends. You can refer to and modify these variables within and outside procedures and functions. They retain their values throughout execution of the script.

Default variable initialization values

The local and global variables you declare are initialized to default values when they are created. The default value for each data type is as follows:

`integer`	0
`real`	0.0
`string`	"" (the null string)
`boolean`	False.
`array`	Each element is initialized to the array-type default.

Local variables are initialized each time the procedure or function begins execution. Global variables are initialized once when the script begins execution.

Procedure and function arguments are like local variables, but they are not initialized to default values like other local variables. They receive their values from the actual arguments.

Labels

The scope of labels you declare inside procedures and functions terminates when the function or procedure that defines them ends. You can refer to these labels only while the procedure or function is executing, and only from within the procedure or function.

The scope of labels you declare outside procedures and functions terminates when the script ends. Procedures and functions cannot reference labels that are not defined within the procedure or function.

4 Interfacing with the Host, Users, and Other Scripts

Interacting with the host 4-2

Communicating with a user 4-6

Invoking other scripts 4-9

Trapping and handling errors 4-11

Interacting with the host

Many of the scripts you develop involve communicating with a host computer. CASL provides a number of language elements you can use to interact with a host. For example, the `wait` statement provides basic data-handling functions while the `watch` statement offers more sophisticated methods for handling data.

In the sections that follow you will see how to use these and other CASL elements to control your script's interaction with the host.

Waiting for a character string

Use the `wait` statement when you need to wait for a specific, unique string of text. The following is an example:

```
wait for "What is your first name?"
```

Note that the string `"What is your first name?"` is enclosed in quotation marks because it is a string constant.

The `wait` statement does not require a complete sentence as shown in the previous example. If just the word `"name?"` is unique at the time the script executes the `wait` statement, you can shorten the statement as follows:

```
wait for "name?"
```

You can have your `wait` statement wait for one of several conditions to occur. For example, if you want to send a carriage return when your script receives either `"more"` or `"press enter"` from the host, write the statement as follows:

```
wait for "more", "press enter" : reply
```

The default wait time for the `wait` statement is forever. You can specify a specific time period for the script to wait, as shown in the following example.

```
reply                                   -- Send CR
wait 2 seconds for "login:"             -- Wait
if timeout then
{
    alert "Host not responding", ok
    end
}
```

In this example, the script waits `2 seconds` for the host to send the `login:` prompt. If a time-out occurs before the prompt appears, the user is alerted and the script ends.

By default, the `wait` statement is not case- or space-sensitive. If your script requires an exact match, you must use the statement's `case` or `space` modifiers or both. There are several other conditions for which a wait statement can wait, including waiting to receive a specific "count" of characters and waiting for the connection to be "quiet." Refer to Chapter 6, "Using the Programming Language," for a complete list of `wait` conditions.

Watching for one of several conditions to occur

Use the `watch/endwatch` construct when you need to wait for any one of several conditions to occur and then take an action based on that condition. The following is an example:

```
watch for
    key 27, "$" : end
    "more:"     : wait 1 second : reply
endwatch
```

In this example, when the `watch` statement is encountered, the script pauses while waiting for one of the 2 conditions to take place. The statement, or statements, to the right of the colon are executed for whichever condition occurs first.

Note that `watch/endwatch` is not a looping construct. If you want to repeat the `watch/endwatch` statements, enclose them in a `while/wend` or a `repeat/until` construct. The following example shows the `while/wend` construct:

```
while online
    watch for
        key 27, "$" : end
        "more:"     : wait 1 second : reply
    endwatch
wend
```

This example is taken from a simple script that automates "reading" electronic mail on a host. The `while/wend` loop is needed because the `more:` prompt will appear multiple times during the reading process.

As specified by the first line of the `watch` construct in the previous example, the script ends if the user presses the ESC key (`key 27`). If `more:` is found, the script waits 1 second and then uses the `reply` statement to send a carriage return to the host. If the dollar sign (`$`) appears, there is no more mail to read, and the script ends.

The `watch` statement, like the `wait` statement, can watch for several different kinds of conditions. Refer to Chapter 6, "Using the Programming Language," for a complete list of the conditions.

Capturing data

Use the `capture` statement to capture and save data that the host displays on the screen. You can use `capture` with either the `wait` or the `watch` statement. The following example shows how to capture data using `wait`:

```
wait 5 seconds for "stock prices for"
if not timeout then
{
   capture "stock.dat"
   wait for "end of listing"
   capture off
}
else print "Never received stock prices."
```

In this example, the script waits 5 seconds for a message that indicates the host is going to send today's stock prices. If a time-out does not occur, the data is captured in a file named `stock.dat` and when the message `"end of listing"` is received, the script turns off the `capture` statement. If a time-out occurs, a message is displayed on the screen.

To make this type of operation more versatile, use the `watch/endwatch` construct inside a `while/wend` loop. This allows the script to wait for both the string that will turn `capture` on and the string that will turn it off all in the same loop. The following is an example:

```
while online
   watch for
      "stock prices for" : capture "stock.dat"
      "end of listing"   : capture off
      key 27             : capture off : end
   endwatch
wend
```

In addition to watching for the 2 character strings, the script in this example is also watching for ESC (`key 27`). If this key is pressed, `capture` is turned off and the script ends.

For more information about the `capture` statement, see Chapter 6, "Using the Programming Language."

Setting and testing time limits

Use the `timeout` system variable to determine if the condition for which you are waiting or watching has occurred within an expected time frame. To use the `timeout` system variable, you must set a time-out value for the `wait` or `watch` condition. Then you can test the `timeout` system variable; it returns `true` if the condition was not satisfied or `false` if it was satisfied.

For example, sometimes a user has to press ENTER a number of times before the host recognizes the response. You can set up a simple routine to handle this situation:

```
repeat
   reply
   wait 1 second for "Login:"
until not timeout
reply userid
end
```

This example shows how to use the `repeat/until` construct to execute the same statements one or more times. When the `repeat/until` condition is satisfied, script execution continues with the statement following the `repeat/until` construct.

In the example, the script uses the `reply` statement without an argument to send only a carriage return character to the host. Then it waits `1 second` for the string `"Login:"` to arrive. If the string does not arrive within the 1-second time frame (`timeout` is `true`), the script repeats the statements in the `repeat/until` construct. If the string arrives within the time frame specified (`timeout` is `false`), the script sends the contents of the system variable `userid` to the host and ends. The `userid` variable must be defined in the user's profile for the session running this script.

Sending a reply to the host

Many of the examples in this section use the `reply` statement to respond to the host computer. The `reply` statement lets you send a string of text to the host. If you use the statement without a text string argument, only a carriage return is sent. You can concatenate more than one string in a `reply` statement by using the plus symbol (+) to join the strings, as shown in the following example:

```
reply userid + " " + password
```

Communicating with a user

In addition to interacting with a host computer, your scripts may also have to communicate with a user. CASL has several language elements specifically designed for interfacing with a user: `print`, `message`, `input`, `alert`, and `dialogbox ... enddialog`. This section describes how you can use these statements to display information for the user and request information from the user.

Displaying information

Use the `print` statement to display information in the session window. You can display constants, variables, or a combination of the two; and you can control such display characteristics as attributes for bright or flashing characters and for color. Note that attributes will work only if the terminal tool, which controls the interface between the script and a terminal, understands what the attributes mean.

The following are examples of simple `print` statements:

```
print "Greetings."

print time(cursecond)

print "The time is " ; time(cursecond)

print "This is all on the ";
print "same line."
```

The first example displays the phrase `Greetings`. The second and third examples display the time. Note that the `print` statement in the third example contains a semicolon. The semicolon causes the text string and the time to be displayed with no space between them.

The fourth example shows how to use the semicolon at the end of a `print` statement to suppress a carriage return. In this example, both `print` statements display text strings that appear on the same line of the screen.

You create a more complex `print` statement when you display words with an attribute. This is shown in the following example:

```
print "This is a ";bright;"bright " ;...
   normal;"idea!"
```

In this example, the `bright` option is used to display the word "bright" using the bright attribute. Note that when an attribute is set, it remains in effect until another attribute is specified. In the example, the `normal` option resets the attribute to normal.

A special character, `^G`, causes the terminal to beep when the `print` statement is executed. The reason for this is that the `print` statement can print ASCII control characters. This attribute is shown in the following example:

```
print "Beep!^G"
```

The `^G` in the example is the ASCII decimal 07 or Bell. Refer to Chapter 2, "Understanding the Basics of CASL," for a list of other ASCII control characters.

The `message` statement allows you to display user-defined messages on the status bar of the session window. The following is an example of a message:

```
message "Logging on -- Please wait"
```

Requesting information

Use the `input` statement to obtain information from the user. The `input` statement suspends the script while waiting for the user to enter data. When the user presses the ENTER key, `input` knows that data entry is complete. The data entered is stored in a specified variable.

The following is an example of how to use the `input` statement:

```
string user_name

print "Please enter your name: " ;
input user_name
print "Hello, "; user_name
```

In the previous example, `user_name` is declared as a string variable. Since the `input` statement does not display a prompt, the `print` statement requests the user to enter a name. After the user enters a name and presses ENTER, the entry is stored in the string variable `user_name`. This variable is then used in the last `print` statement to display the name that was entered.

The `alert` and `dialogbox ... enddialog` statements allow you to define Windows or Macintosh-style dialog boxes for text input.

The `alert` statement displays a simple dialog box in which the user can enter text or respond by choosing a pushbutton. The `dialogbox/enddialog` construct allows you to create more sophisticated dialog boxes, which can contain pushbuttons, text, edit boxes, radio buttons, check boxes, list boxes, and so on.

The following is an example of an `alert` statement that displays a message:

```
alert "File not found", "Try again", cancel, ok
```

In this example, the message `File not found` is displayed in the alert box. The user can choose either Try Again, Cancel, or OK to exit the alert box.

Refer to Chapter 6, "Using the Programming Language," for additional information about the `print`, `message`, `input`, `alert`, and `dialogbox ... enddialog` statements.

Invoking other scripts

With CASL, you can invoke, or start, another script from your script. Depending on your programming requirements, your script can terminate and pass control (chain) to the other script; or your script can use the `do` statement to call the other script as a child script.

Chaining to another script

To pass control to another script without returning control to your script, use the `chain` statement. For example, to pass control to a script called SCRIPT2, write the `chain` statement as follows:

```
chain "SCRIPT2"
```

Note: Any statements that follow the `chain` statement are not executed. ■

Calling another script

To call another script as a child script, use the `do` statement. When you use this statement, the child script returns control to the parent script when the child script has completed. The following is an example of the `do` statement:

```
do "cvtsrc"
```

Passing arguments

To pass arguments to the invoked script, add the arguments to the `chain` or `do` statement after the name of the script. In the following `chain` statement, the argument `CSERVE` is passed to SCRIPT2:

```
chain "SCRIPT2 CSERVE"
```

To retrieve the arguments in the invoked script, use the `arg` function. Use `arg` with no arguments (or an argument of zero) to retrieve the arguments as one long string. Use `arg(1)` through `arg(`*n*`)` to retrieve each individual argument.

Exchanging variables

If you use the `do` statement to invoke another script, the scripts can exchange variable information. To pass a variable between scripts, declare the variable as `public` in the invoking script and as `external` in the invoked script.

In the following example, the invoking script, SCRIPT1, declares the string `myname` as `public`, invokes `SCRIPT2`, prints a message when SCRIPT2 returns control, and ends.

```
public string myname
do "SCRIPT2"
print "My name is " + myname
end
```

In the next example, SCRIPT2, which was invoked by SCRIPT1, declares the string variable `myname` as `external`, assigns a value to `myname`, and returns control to SCRIPT1. Note that the value SCRIPT2 assigns to `myname` is what SCRIPT1 prints when it regains control (see the first example).

```
external string myname
myname = "Bert"
end
```

The message that SCRIPT1 displays on the screen is as follows:

```
My name is Bert
```

Note: You cannot exchange data with another script if you use the `chain` statement to invoke the script. Also, if you are using public and external variables, you must declare the variable as `public` in the parent script. ■

Trapping and handling errors

Error trapping makes a script capable of handling almost any situation, and it is essential in scripts that are interfacing with other resources. With error trapping, you can control many different situations; for example, you can set up recovery procedures if a file transfer or file input/output operation fails. In the following sections, you will see how to enable error trapping, determine if an error occurred, check the type of error, and check the error number. You can also find a sample script that shows how to trap and handle errors.

Enabling error trapping

Use the `trap` compiler directive to enable and disable error trapping in your script. The default setting for this directive is `trap off`. If `trap` is `off`, a dialog box is automatically displayed and the script terminates whenever a fatal error occurs. If `trap` is `on`, the dialog box is not displayed; rather, the script continues executing.

In general, it is best to turn trapping on just prior to a statement that may generate an error and then turn it off after testing for the error. Be sure to check the error-trapping function `error`, and the system variables `errclass`, and `errno` just after the statement executes; otherwise, you may lose the error information if a subsequent statement resets the error function and variables. (See the following sections for an explanation of these elements.)

Testing if an error occurred

Use the `error` function to test if an error occurred. This function returns true if an error occurs or false if no error occurs. When you test the function, its value is reset to zero. If you want to continue to trap errors throughout the execution of the script, you must test (reset) the `error` function each time an error occurs.

Checking the type of error

Use the `errclass` system variable to check the type of error that occurred. This variable contains zero if no error occurs; if an error does occur, it contains an integer value that reflects the type of error. This variable is not reset when you check its value. The value remains unchanged until another error occurs. For information on the `errclass` values you may encounter, refer to Appendix C, "Error Return Codes."

Checking the error number

Use the `errno` system variable to check the number of the error that occurred. The error number is associated with the type of error that is returned by the `errclass` variable. For example, the return code 13-08 represents the `errclass` value 13 and the `errno` value 08; this type of error is a file I/O read error. (For additional information, see Appendix C, "Error Return Codes.")

If no error occurs, the `errno` variable contains zero. This variable is not reset when you check its value; the value remains unchanged until a different error occurs.

When setting up your script to trap and handle errors, follow these guidelines, in the order shown:

- Set `trap on` right before a statement that could generate an error condition (for example, a statement that sends files to the host). Note that setting `trap on` suppresses error message display.
- Set `trap off` immediately after the statement executes.
- Check the `error` function after setting `trap off`.
- If an error occurs (`error` is true), check the `errclass` and `errno` system variables to determine the error type and number.

The following sample script illustrates how to use CASL's error trapping capabilities. The script's purpose is to send a file to the host. If the file transfer is successful, the script ends. If, for any reason, the file transfer does not complete successfully, the script sounds an alarm and prints an error message.

```
/* Script to send a file. */
string fname
fname = "*.exe"
trap on                -- turn on error trapping
send fname             -- send the file
trap off               -- turn off error trapping
if error then
{
   alarm
   print "Send failed. Error: "; + ...
      errclass; "-"; errno
}
end
```

This script is very simple and is shown here only to illustrate how you can use `trap`, `error`, `errclass`, and `errno` to handle an error condition. Ideally, your error-handling should be more comprehensive. For example, if the script is unattended, error handling should either attempt to send the file again or hang up and retry later, depending on the error type. If the script is attended, error handling might print a message that informs the user of the error and instructs the user to correct the problem and retry the file transfer.

It is not always necessary to determine the values in `errclass` and `errno`; sometimes it is sufficient just to know that an error occurred (by checking `error`). How you use error trapping and to what extent depends on what your script needs to accomplish.

Refer to Chapter 6, "Using the Programming Language," for more information on the `trap` compiler directive, the `error` function, and the `errclass`, and `errno` system variables.

5 Introducing the Programming Language

Functional purpose of CASL elements 5-2

Functional purpose of CASL elements

This chapter and Chapter 6, "Using the Programming Language," provide reference information to help you use the CASL elements. This chapter contains a quick reference to all of the elements. A detailed description of the elements and examples showing how to use them in your scripts are covered in Chapter 6.

The CASL elements in this chapter are grouped according to their functional purpose, for example, session management, program flow control, file input/output operations, and so on. Some elements may appear more than once if they have more than one purpose. A brief description of the element is also included. Each description ends with an element identifier as follows:

F	Function
S	Statement
V	Variable (system and module)
C	Constant
D	Declaration (procedure and function)
CD	Compiler directive
DH	Data handler (DDE) ■

Win

Capture and upload control

The language elements that control the capture of data and the upload of data to the host are as follows:

`add`	Adds text to the capture file. (S)
`blankex`	Controls the way a blank line is represented during uploads. (V)
`capchars`	Returns the number of characters captured. (F)
`capfile`	Returns the name of the capture file. (F)
`capture`	Turns capture on and off. (S)
`cmode`	Specifies a capture method. (V)

`cwait`	Controls the inter-character delay during uploads. (S)
`dirfil`	Defines the directory used for transfers and captures. (V)
`downloaddir`	Defines a different directory to be used for transfers and captures. (V)
`grab`	Writes window data to the capture file. (S)
`linedelim`	Sets the string to send at the end of each line. (S)
`linetime`	Sets the maximum time to wait between each line. (S)
`lwait`	Controls the inter-line delay during uploads. (S)
`tabex`	Defines the tab expansion during uploads. (V)
`upload`	Initiates a text file upload. (S)

Date and time operations

The following language elements help you determine the date and time:

`curday`	Returns the current day of the month. (F)
`curhour`	Returns the current hour. (F)
`curminute`	Returns the current minute. (F)
`curmonth`	Returns the number of the current month. (F)
`cursecond`	Returns the current second. (F)
`curyear`	Returns the current year. (F)
`date`	Returns today's date as a string. (F)
`hms`	Returns a string in hours, minutes, and seconds format. (F)

`secno`	Returns the number of seconds since midnight. (F)
`time`	Returns the current time as a string. (F)
`weekday`	Returns the number of the day of the week (0–6). (F)

Win DDE interface

The language elements that allow interaction with other applications using Dynamic Data Exchange are as follows:

`ddeack`	Sends an acknowledgment to a `ddeadvise` request. (S)
`ddeadvise`	Requests notification of all changes to a specified data item. (S)
`ddeadvisedatahandler`	Enables the event handler that will handle `ddeadvise` message events. (DH)
`ddeexecute`	Requests that another application execute a command. (S)
`ddeinitiate`	Opens a DDE conversation with another application. (S)
`ddenak`	Sends a negative acknowledgment to a `ddeadvise` request. (S)
`ddepoke`	Sends a string of data to the application at the other end of a DDE conversation. (S)
`dderequest`	Requests a value from another application. (S)
`ddestatus`	Returns the status of the DDE conversation. (F)
`ddeterminate`	Terminates a DDE conversation. (S)
`ddeunadvise`	Cancels a previous `ddeadvise` request. (S) ■

Device interaction

The language elements that control interaction with a communications device are as follows:

`connectreliable`	Contains the modem result string that indicates a reliable, or error-free, connection. (V)
`dialmodifier`	Defines the dialing modifier string used to command the modem to dial. (V)

Note: Refer to Chapter 7, "Working with Terminal, Connection, and File Transfer Tools," for an explanation of the connection tool. ■

Error control

The following language elements help you control error conditions in your scripts:

`errclass`	Indicates the class of the last error. (V)
`errno`	Indicates the type of the last error. (V)
`error`	Indicates the occurrence of an error. (F)
`trap`	Turns error trapping on and off. (CD)

File input/output operations

The following language elements provide file input and output capabilities:

	`backups`	Determines what is done with duplicate files after a file transfer. (V)
	`capture`	Captures incoming text to a file. (S)
	`chdir`	Changes to a different disk directory. (S)
Win	`chmod`	Changes file attributes. (S) ■
	`close`	Closes a disk file. (S)
	`copy`	Copies a file or group of files. (S)
	`curdir`	Returns the current disk directory. (F)
Win	`curdrive`	Returns the current disk drive. (F) ■

	definput	Contains the default input file number. (V)
	defoutput	Contains the default output file number. (V)
	delete	Deletes disk files. (S)
Win	drive	Sets the current disk drive. (S) ■
	eof	Returns true if end-of-file is reached. (F)
	eol	Returns true if end-of-line is reached. (F)
	exists	Returns true if a file exists. (F)
	fileattr	Returns the attributes of a file. (F)
	filedate	Returns the file date stamp. (F)
	filefind	Locates files in the directory. (F)
	filesize	Returns the file size. (F)
	filetime	Returns the file time stamp. (F)
Win	fncheck	Checks the validity of a file name. (F) ■
Win	fnstrip	Returns specified portions of a file name. (F) ■
	freefile	Returns the next available file number. (F)
	get	Reads characters from a random access file. (S)
	kermit	Sends a command to the Kermit Command Processor. (S)
	loc	Returns a file pointer position. (F)
	mkdir	Creates a new directory. (S)
	open	Opens a disk file. (S)
	put	Writes records to a random disk file. (S)
	read	Reads text fields from a file. (S)
	read line	Reads text lines from a file. (S)
	receive	Initiates a file transfer. (S)

`rename`	Renames disk files. (S)
`rmdir`	Removes a disk directory. (S)
`seek`	Moves a file pointer to a specified position. (S)
`send`	Initiates a file transfer to a remote computer. (S)
`upload`	Uploads an ASCII text file to the host. (S)
`write`	Writes text fields to a file. (S)
`write line`	Writes text lines to a file. (S)

Host interaction

The language elements that let you interact with another computer are as follows:

`breaklen`	Specifies the length of a break signal. (V)
`display`	Turns a terminal display on and off. (V)
`match`	Specifies the string found by the last `wait` or `watch` statement. (V)
`nextchar`	Returns the next character from a communications device. (F)
`nextline`	Returns the next line, delimited by a carriage return, from the communications device. (F/S)
`online`	Returns true if a session is on line. (F)
`press`	Sends a series of keystrokes to the terminal module. (S)
`reply`	Sends a string of text to the communications device. (S)
`sendbreak`	Sets the length of a break signal. (S)

`track`	Watches for string patterns or keystrokes while on line. (S)
`wait`	Waits for a string of text from the communications device or for a keystroke. (S)
`watch/endwatch`	Watches for one of several conditions to occur. (S)

Mathematical operations

The following language elements perform mathematical operations:

`abs`	Returns the absolute value of a number. (F)
`cksum`	Returns the checksum of a string. (F)
`crc`	Returns the CRC of a string. (F)
`intval`	Returns the integer value of a string. (F)
`max`	Returns the larger of two values. (F)
`min`	Returns the smaller of two values. (F)
`mkint`	Converts numeric strings to integers. (F)
`val`	Returns the real (floating point) value of a string. (F)

Printer control

The language elements that control how data is printed are as follows:

`footer`	Defines the footer used when printing. (V)
`header`	Defines the header used when printing. (V)
`lprint`	Sends a string of text to the printer. (S)
`printer`	Indicates whether to send screen output to the printer. (V)

Program flow control

The following language elements provide program flow control in your scripts:

`case/endcase`	Performs statements based on the value of a specified expression. (S)
`chain`	Passes control to another script. (S)
`do`	Starts another script and waits until it returns control. (S)
`end`	Ends a script. (S)
`exit`	Exits a procedure. (S)
`for/next`	Performs a series of statements a specified number of times, usually while changing the value of a variable. (S)
`freetrack`	Returns the value of the lowest unused track number for the current session. (F)
`func/endfunc`	A function declaration. (D)
`gosub/return`	Transfers program control to a subroutine. (S)
`goto`	Transfers program control to a label or expression. (S)
`halt`	Stops a script and its related parent and child scripts. (S)
`if/then/else`	Controls program flow based on the value of an expression. (S)
`label`	Denotes a named reference point in a script. (S)
`new`	Begins a new communications session. (S)
`perform`	Calls a procedure. (S)
`proc/endproc`	A procedure declaration. (D)
`quit`	Closes a session window. (S)

`repeat/until`	Repeats a statement or series of statements until a specified condition is true. (S)
`return`	Returns a value from a function. (S)
`terminate`	Terminates the Crosstalk application. (S)
`timeout`	Returns the status of the most recent `wait` or `watch` statement. (V)
`trace`	Turns tracing on and off. (S)
`track`	Watches for string patterns or keystrokes while on line. (S)
`wait`	Waits for a string of text from the communications device or for a keystroke. (S)
`watch/endwatch`	Watches for one of several conditions to occur. (S)
`while/wend`	Performs a statement or group of statements as long as a specified condition is true. (S)

Script and session management

The language elements that help you manage sessions and scripts are as follows:

`activate`	Activates the Crosstalk window by moving the focus to it. (S)
`activatesession`	Makes the specified session active. (S)
`active`	Makes Crosstalk the active application. (F)
`activesession`	Indicates the session that is active. (F)
`assume`	Controls the way the CASL compiler handles module variables for the Connection, Terminal, and File Transfer tools. (S)
`bye`	Disconnects the current session. (S)

`call`	Initiates a connection for a communications session. (S)
`chain`	Passes control to another script. (S)
`compile`	Compiles a script. (S)
`description`	Defines a session. (V)
`device`	Specifies a connection device. (V)
`dirfil`	Defines the default directory used for transfers and captures. (V)
`do`	Starts another script and waits for it to return control. (S)
`downloaddir`	Defines a different directory to be used for transfers and captures. (V)
`genlabels`	Specifies whether to include or exclude label information in a compiled script. (CD)
`genlines`	Specifies whether to include or exclude line information in a compiled script. (CD)
`go`	Initiates a connection to a communications device. (S)
`include`	Includes an external file in a compiled script. (CD)
`inscript`	Checks for labels in a script. (F)
`keys`	Specifies the Keymap file for the current session. (V)
`load`	Starts a session. (S)
`name`	Contains the name of the current session. (F)
`netid`	Contains the network identifier for a session. (V)
`number`	Contains the phone number for the current session. (V)
`ontime`	Indicates how long a session has been on line. (F)

`password`	Contains the password for the current session. (V)
`patience`	Specifies the amount of time to wait for an answer from the host. (V)
`protocol`	Specifies a file transfer protocol. (V)
`quit`	Closes a session window. (S)
`redialcount`	Specifies the number of redial attempts. (V)
`redialwait`	Specifies how long to wait before attempting to redial. (V)
`run`	Starts another application. (S)
`save`	Saves the current session parameters. (S)
`script`	Specifies the name of the script file to use for the current session. (V)
`scriptdesc`	Defines a script description. (CD)
`session`	Returns the session number of the current session. (F)
`sessname`	Returns the name of the session identified by a specified session number. (F)
`sessno`	Returns the session number of a specified session. (F)
`startup`	Contains the name of the script to run at start-up. (V)
`terminal`	Specifies the terminal emulation to use. (V)
`terminate`	Terminates the Crosstalk application. (S)
`trace`	Turns tracing on and off. (S)
`userid`	Contains the user account name for a session. (V)

String operations

The following language elements perform string operations:

`arg`	Returns command line arguments. (F)
`bitstrip`	Removes bits from strings. (F)
`count`	Returns the number of occurrences of one string within another string. (F)
`dehex`	Converts ASCII strings in hexadecimal format to binary. (F)
`delete`	Returns a string with characters removed. (F)
`destore`	Converts strings of printable ASCII characters back to embedded control-character form. (F)
`detext`	Converts 7-bit ASCII character strings to binary. (F)
`enhex`	Converts a binary string to a string of ASCII characters in hexadecimal format. (F)
`enstore`	Converts strings with embedded control characters into strings of printable ASCII characters. (F)
`entext`	Converts a string of binary data to a string of 7-bit ASCII characters. (F)
`extract`	Extracts characters from a string. (F)
`hex`	Converts an integer to a hexadecimal string. (F)
`hms`	Returns a string in hours, minutes, and seconds format. (F)
`inject`	Changes some characters in a string. (F)
`insert`	Adds characters to a string. (F)
`instr`	Looks for a substring in a string. (F)
`intval`	Returns the integer value of a string. (F)
`left`	Returns the left portion of a string. (F)
`length`	Returns the length of a string. (F)

`lowcase`	Changes a string to all lowercase characters. (F)
`mid`	Returns a middle portion of a string. (F)
`mkstr`	Converts an integer to a string. (F)
`null`	Returns true if a string has zero length. (F)
`pack`	Removes duplicate characters from a string. (F)
`pad`	Adds extra characters to a string. (F)
`quote`	Returns a string enclosed in quotation marks. (F)
`right`	Returns the right portion of a string. (F)
`slice`	Breaks out portions of a string. (F)
`str`	Converts a number to string format. (F)
`strip`	Returns a string with certain characters removed. (F)
`subst`	Returns a string with certain characters changed. (F)
`upcase`	Changes a string to all uppercase characters. (F)
`val`	Returns the real (floating point) value of a string. (F)
`winstring`	Reads a string from a window. (F)

Type conversion operations

The following language elements let you convert data from one type to another:

`asc`	Returns the ASCII value of a string. (F)
`binary`	Converts a string to a binary number. (F)
`bitstrip`	Strips bits from strings. (F)
`chr`	Returns a single-character string for an ASCII value. (F)

`class`	Returns the class type of a single-character string. (F)
`dehex`	Converts ASCII strings in hexadecimal format to binary. (F)
`detext`	Converts 7-bit ASCII character strings to binary. (F)
`enhex`	Converts a binary string to a string of ASCII characters in hexadecimal format. (F)
`entext`	Converts a string of binary data to a string of 7-bit ASCII characters. (F)
`hex`	Converts an integer to a hexadecimal string. (F)
`intval`	Returns the integer value of a string. (F)
`mkint`	Converts numeric strings to integers. (F)
`mkstr`	Converts an integer to a string. (F)
`octal`	Converts a decimal integer to an octal integer. (F)
`str`	Converts a number to string format. (F)
`val`	Returns the real (floating point) value of a string. (F)

Window control

The following language elements control the window size and how data is input and displayed in a window:

`activate`	Activates the Crosstalk window by moving the focus to it. (S)
`activatesession`	Makes the specified session active. (S)
`active`	Makes Crosstalk the active application. (F)
`activesession`	Indicates the session that is active. (F)

	`alert`	Creates simple dialog boxes for display on the screen. (S)
	`choice`	Contains the value of the pushbutton that dismissed a dialog box. (V)
	`clear`	Clears a window. (S)
	`dialogbox/enddialog`	Creates more complex dialog boxes for display on the screen. (S)
	`hide`	Reduces a session window to an icon. (S)
	`hideallquickpads`	Hides all of the QuickPads. (S)
	`hidequickpad`	Hides a QuickPad™. (S)
	`input`	Accepts input from the screen. (S)
	`loadquickpad`	Activates a QuickPad. (S)
	`maximize`	Enlarges the Crosstalk window to full-screen size. (S)
	`message`	Displays a message in the information line on the screen. (S)
	`minimize`	Reduces the Crosstalk window to an icon. (S)
Win	`move`	Moves the Crosstalk window to a new location on the screen. (S) ■
	`print`	Displays information on the screen. (S)
	`restore`	Restores the Crosstalk window to its original size. (S)
	`show`	Redisplays a session window. (S)
	`showallquickpads`	Displays all of the QuickPads. (S)
	`showquickpad`	Displays a QuickPad. (S)
Win	`size`	Changes the size of a window. (S) ■
	`tabwidth`	Specifies the number of spaces a tab character moves the cursor. (V)

`unloadallquickpads`	Closes all of the QuickPads. (S)
`unloadquickpad`	Closes a QuickPad. (S)
`winchar`	Reads a character from a window. (F)
`winsizex`	Returns the horizontal size of a window. (F)
`winsizey`	Returns the vertical size of a window. (F)
`winstring`	Reads a character string from a window. (F)
`xpos`	Returns the horizontal location of the cursor. (F)
`ypos`	Returns the vertical location of the cursor. (F)
`zoom`	Enlarges a session window to the size of the Crosstalk application window. (S)

Miscellaneous elements

The following are general purpose language elements:

`alarm`	Sounds an alarm at the terminal. (S)
`dosversion`	Returns the operating system version number. (F) ■
`environ`	Returns environment variables. (F) ■
`false`	Sets a variable to logical false. (C)
`freemem`	Returns the amount of available memory. (F)
`inkey`	Returns the value of a keystroke. (F)
`off`	Sets an item to logical false. (C)
`on`	Sets an item to logical true. (C)
`pop`	Discards a return address from the stack. (S)
`review`	Defines the size of the review buffer. (V)
`stroke`	Waits for the next keystroke from the keyboard. (F)

`systime`	Indicates how long the current session has been active. (F)
`true`	Sets a variable to logical true. (C)
`version`	Returns the Crosstalk version number. (F)
Win `winversion`	Returns the Windows version number. (F) ■

CASL language elements have a specific format and use. To learn how to structure and implement each element, turn to Chapter 6, "Using the Programming Language."

6 USING THE PROGRAMMING LANGUAGE

Information provided for CASL elements 6-2

Language elements 6-3

Information provided for CASL elements

The following items are described for each CASL language element:

Language element name	The element name is shown in large bold typeface below a line that extends the width of the page. A paragraph that describes the purpose of the element follows the name.
Format	This section shows the format for the language element. Where applicable, components are explained in more detail. Compatibility information is also provided where appropriate. For detailed information about compatibility issues, refer to Chapter 8, "Compatibility Issues." **Note:** For a description of the notation used in the format, see Chapter 2, "Understanding the Basics of CASL." ■
Example	In this section, you find an illustration of how you can use the language element in your script. An explanation of the example follows the illustration.

abs (function)

Use abs to get the absolute value of a number.

Format

```
x = abs(<expression>)
```

expression must be a real or signed integer. The result returned by the abs function is always a positive number.

Examples

```
positive_number = abs(negative_number)
```

In this example, abs assigns the absolute value of the contents of negative_number to the variable called positive_number.

```
if abs(net_worth) > 5 then alarm
```

In this example, the script sounds an alarm if the absolute value of the net_worth variable is greater than 5.

activate (statement)

Use `activate` to make the Crosstalk window the active window.

Format

```
activate
```

When you use this statement, the focus is moved from the active window to the Crosstalk window, making the Crosstalk window the active one.

This function is not applicable for Crosstalk Mark 4.

For related information, see the `activatesession` statement and the `active` and `activesession` functions.

Example

```
activate
```

activatesession (statement)

Use `activatesession` to make the specified session active.

Format

```
activatesession <sessionid>
```

When you use this statement, the session identified by *sessionid* becomes active.

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support this function.

For related information, see the `activate` statement and the `active` and `activesession` functions.

Examples

```
activatesession sessA
```

In this example, session A becomes active.

```
activatesession sessno("ABBS")
```

In this example, `activatesession` activates the session named ABBS whose session number is returned by the `sessno` function.

active (function)

Use `active` to check whether Crosstalk is the active window.

Format

```
x = active
```

This function returns `true` if Crosstalk is the active window. (The active window is the application that receives input from the keyboard.) It returns `false` if another application has the focus. Note that you can store the return value in an integer even though it is a boolean data type.

For Crosstalk Mark 4, `active` returns an integer indicating the currently active communications session.

For related information, see the `activate` and `activatesession` statements and the `activesession` function.

Example

```
if active then reply "I'm it!"
```

In this example, a `reply` is sent to the connected system if `active` is true.

activesession (function)

Use `activesession` to check which session is active.

Format

```
x = activesession
```

This function returns the number of the active session.

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support this function.

For related information, see the `activate` and `activatesession` statements and the `active` function.

Example

```
x = activesession
if sessname(x) = "CSERVE" then
{
   print "CSERVE is active."
}
```

In this example, the script displays a message if the session returned by the `sessname` function is CSERVE.

add (statement)

Use add to append text to the capture file.

Format

```
add [<string>] [{ , | ; } [<string>]] ... [ ; ]
```

string is a string or a string expression that should be added to the capture file. It is added to the file just as if it had been received at the communications port.

Use the comma (,) if you want a tab character between strings. If two or more commas are together, two or more tabs are added. For example, if you use 3 commas in succession, 3 tab characters are added. Use a semicolon (;) to suppress the tabs.

The add statement normally adds a carriage-return/line-feed (CR/LF) character after the last string. To suppress the CR/LF, add a semicolon after the last string.

For related information, see the capture statement.

Examples

```
add "This was captured on " + date + ...
    " at " + time(cursecond)
```

In this example, the script adds the message This was captured on, the current date, the word "at", the current time, and a CR/LF to the capture file.

```
add xferfile, xferdate, xferwho;
```

In this example, the script adds the contents of the user-defined variables xferfile, xferdate, and xferwho, separated by tabs, to the capture file. The CR/LF is suppressed because the statement ends with a semicolon.

alarm (statement)

Use `alarm` to make the terminal sound an alarm.

Format

`alarm [<integer>]`

This function is useful for getting the user's attention.

integer can be any integer between 0 and 12; values out of range are ignored. "0" is the default alarm used when no argument is specified.

Table 6-1 shows possible *integer* values and their corresponding alarm sound.

Table 6-1. Alarm sounds

Integer value	Sound description
0	Short beep
1	Close Encounters of the Third Kind
2	3 beeps
3	DK's music
4	4-note "toot"
5	Beethoven's Fifth
6	"Twilight Zone"
7	Dirge
8	"The Deaconess of Detroit"
9	"Popeye the Sailor Man"
10	Fanfare
11	"Up" sound
12	"Down" sound

Versions of Crosstalk for Windows older than 2.0 do not allow an argument and beep only once.

Examples

```
alarm 1
```

In this example, the terminal plays the "Close Encounters of the Third Kind" theme.

```
if not exists("BBS.DAT") then alarm
```

In this example, the `exists` function is used to determine the existence of a file. If the file does not exist, the script sounds an alarm.

```
for i = 0 to 12
   print "alarm "; i
   alarm i
   wait 1 second
next
```

In this example, the terminal sounds all of the alarms, with a pause of 1 second between each alarm.

alert (statement)

Use `alert` to display a dialog box that allows choices to be made.

Format

```
alert <string>, <button1> [, <button2> ...
   [, <button3> [, <button4>]]] [, <str_var>]
```

The `alert` statement displays a dialog box that prompts the user for input, or notifies the user of some important occurrence.

A text message defined by *string* is centered in the dialog box. The defined pushbuttons are displayed along the bottom of the dialog box. *button1* through *button4* is the text to display in the pushbutton. You can use `ok` and `cancel`, which are predefined keywords, as pushbutton arguments; you do not need to enclose them in quotation marks. The maximum length of a pushbutton name is 10 characters. Pushbuttons are displayed from left to right.

If you use the `ok` keyword, `alert` creates an OK pushbutton in the dialog box and associates the ENTER key with this pushbutton. If you use the `cancel` keyword, `alert` creates a Cancel pushbutton in the dialog box and associates the ESC key with this pushbutton.

str_var is a previously defined string variable that causes `alert` to display an edit box in which the user can enter text. The edit box appears between the text message string and the pushbuttons in the dialog box.

You can examine the variables that display or store user information after the `alert` statement has executed. The system variable, `choice`, contains a value between 1 and 4 that corresponds to the pushbutton used to exit the dialog box. For example, if *button1* is chosen, `choice` is set to integer 1. Note that *str_var* is not updated if the Cancel pushbutton is used to exit the dialog box.

Crosstalk normally makes the first letter of the pushbutton name an accelerator. You can define a different accelerator by placing an ampersand (&) ahead of the desired letter. If you use variables for the pushbutton names, make sure the OK and Cancel pushbuttons are last; if the last item is a variable, it is used for a text box.

Crosstalk Mark 4 uses the `alert` command to modify the attributes of a text window. Crosstalk for Windows and Crosstalk for Macintosh do not implement text windows; therefore, these applications use this statement in a different way, as explained earlier.

For related information, see the `dialogbox ... enddialog` statement.

Examples

```
string username

alert "Please enter your name:", ok, username
alert "You entered: " + username, ok
```

In this example, the script displays a dialog box that prompts the user to enter a name. The name that is entered is stored in the variable `username`. A second dialog box displays the contents of `username`.

```
if not exists(filename) then
{
   alert "File not found", "Try again", ok, cancel
   if choice = 1 then goto get_fname
}
```

In this example, the script displays a dialog box that tells the user an invalid file name has been entered. If the user clicks the "Try again" pushbutton, the script branches to its `get_fname` label.

arg (function)

Use arg to check the command-line argument(s) at script invocation.

Format

```
x$ = arg[(<integer>)]
```

arg with no arguments (or an argument of zero) returns all of the arguments that follow the name of a script in the chain or do statement. It can also return everything that was entered in the "Script Arguments" edit box on the Run dialog box, which is accessed from Crosstalk's Action pull-down and in the arguments edit box for defining logon scripts for the session.

arg(1) through arg(n) return the individual elements of the argument, as separated by commas.

For related information see the chain and do statements.

Examples

```
script1.xws
   do "script2", "barkley"
script2.xws:
   fname = arg(1)
   if arg(1) = "barkley" then ...
```

In this example, the first script uses the do statement with the argument barkley to start the second script as a child script. The second script assigns the value in arg(1) to the user variable fname. Then it tests whether the first argument is barkley.

```
menu.xws
   do "LOGIN", "myuserid", "mypassword"

login.xws
   reply arg(1)
   wait for "password:"
   reply arg(2)
```

In this example, the `do` statement is used to run the script file `LOGIN`. `LOGIN` reads its arguments and sends them to the host with the `reply` statement.

asc (function)

Use asc to convert the first character of a string to its corresponding ASCII value.

Format

```
x = asc(<string>)
```

string can be a string constant or expression of any length. When the statement is executed, x contains the ASCII value of the first character in the string. If *string* is not null, the value returned is in the range of 0–255. If *string* is null, (has no length), asc returns a -1.

Examples

```
sixty_five = asc("A")
```

In this example, asc returns the ASCII value of the character "A" in sixty_five.

```
seventy = asc("For pity's sake")
```

In this example, asc returns the value of the character "F," which is the first character of the string "For pity's sake," in the variable seventy.

```
x = asc(mid(thestring, 2, 1))
```

In this example, asc converts the second character of thestring and returns the result in x.

assume (statement)

Use assume to control the way the CASL compiler handles module variables for the Connection, Terminal, and File Transfer tools.

Format

```
assume <module> <filename> ...
   [, <module> <filename>]
```

The Connection, Terminal, and File Transfer tool module variables are not part of Crosstalk's "vocabulary" unless the tools are loaded. The assume statement tells the compiler which tools will be loaded.

The module variables that are a part of the assume statement are available only when the script is compiled. To make the variables available at run time, the specified tool(s) must be loaded for the session running the script.

Valid *module* types are device, protocol, and terminal. *filename*, which must be enclosed in quotation marks, is the name of the tool file you want to be active while the script is compiled.

You can specify multiple tools with one assume statement; however, you should assume them only when the script needs them.

Versions of Crosstalk for Windows older than 2.0 do not support this statement.

Example

```
assume device "DCAMODEM"
```

In this example, the script tells the compiler to assume the tool type device with the name DCAMODEM.

backups (module variable)

Use backups to determine whether to keep or discard duplicate files during file transfers.

Format

```
backups = {on | off}
```

If backups is on and an existing file is received or edited, the old file is renamed with a .BAK extension. If a backup file already exists, it is deleted.

If backups is off and an existing file is received or edited, the old copy of the file is deleted.

Example

```
backups = off
```

In this example, backups is turned off.

binary (function)

Use `binary` to convert an integer to a string, in binary format.

Format

```
x$ = binary(<integer>)
```

The `binary` function returns a binary string that represents the value of *integer*. The string can be 8, 16, or 32 bytes long, depending on the value of *integer*. Integer values and their corresponding binary string lengths are shown in Table 6-2.

Table 6-2. Integer values and their binary string lengths

Integer value	Binary string length
0–255	8
256–65,535	16
65,536–2,147,483,647	32

Example

```
bin_num = binary(some_num)
```

In this example, the value of the variable `some_num` is converted to its binary form, and the new value is stored in the variable `bin_num`.

bitstrip (function)

Use bitstrip to strip certain bits from a string.

Format

```
x$ = bitstrip(<string> [, <mask>])
```

bitstrip produces a new string that is the result of performing a bitwise and of each character in *string* with *mask*. Refer to Chapter 2, "Understanding the Basics of CASL," for an explanation of the bitwise and operation.

mask is an integer bitmap value that defaults to 127 (07Fh), thus stripping the high order bit from each byte in *string*. Some word processors, such as WordStar™, set the high bit in certain characters to indicate various conditions such as special formatting. Stripping the high bit makes such files readable, but it is not a replacement for a true conversion program. A mask of 05Fh (95 decimal) converts lowercase letters to uppercase, but it also changes other characters.

Because *mask* is a bitmap, it must be in the range of 0–255 (decimal); values in the range of 0–127 are the most useful.

For related information, see the lowcase and upcase functions.

Examples

```
readable_string = bitstrip(WordStar_line)
```

In this example, bitstrip strips the high-order bit of each byte of the string WordStar_line and returns the result in readable_string.

```
reply bitstrip(WordStar_line)
```

In this example, bitstrip strips the high-order bit of of each byte of the string WordStar_line and the result is sent to the host with the reply statement.

```
all_upcase = bitstrip("abc", 5Fh)
```

In this example, the letters "abc" are converted to "ABC."

blankex (system variable)

Use `blankex` to substitute a string for a blank line during text uploads.

Format

`blankex = <string>`

Many information services interpret a blank line sent by an on-line user to mean "end of transmission." An example of this is the CompuServe® Forum software, which requires that you enter a period (.) to place a blank line in a message. To substitute a string for a blank line, use the `blankex` system variable.

The most likely character to use for `blankex` is a space, but some services will interpret even that to be a blank line. For those services, use a period or other character.

Examples

```
blankex = "."
```

The variable `blankex` is set to a period.

```
blankex = " "
```

The variable `blankex` is set to a single space.

breaklen (module variable)

Use `breaklen` to set the length of a break signal.

Format

`breaklen = <integer>`

This variable sets the duration of the break signal sent to the host. *integer* is in milliseconds and the range is 10 through 5,000.

For related information, see the `sendbreak` statement.

Example

```
assume device "DCAMODEM"
breaklen = 100
```

In this example, the script sets the break length to 100 milliseconds (.1 seconds).

bye (statement)

Use `bye` to end a connection (hang up).

Format

```
bye
```

This statement immediately disconnects the current communications session and also disconnects the modem connection.

For related information, see the `quit` statement.

Example

```
wait for "Logged off" : bye
```

In this example, the script waits for the phrase "`Logged off`" and then disconnects the session and the modem connection.

call (statement)

Use `call` to load new settings into the current session and then establish a connection.

Format

`call <string>`

The `call` statement loads new settings from the session file named in *string*. If the session file does not exist, an error occurs.

If you do not include a path, the search is limited to the current directory.

Versions of Crosstalk for Windows older than 2.0 prompt the user for a string when no argument is specified. This statement now displays an error. Also, the Crosstalk Mark 4 version of the `call` statement allows arguments to the start-up script for the specified session. This is not supported for the Windows or Macintosh products.

For related information, see the `bye`, `load`, and `quit` statements.

Examples

```
label DoAgain
call "CSERVE"
if not online then goto DoAgain
```

In this example, the script loads new settings from a session file called CSERVE and attempts to establish a connection. If the session is not on line, the `goto` statement branches to the label `DoAgain`.

```
card_name$="CompuServe"
call card_name$
```

In this example, the variable `card_name` is set to the session name `"CompuServe"` and then it is started.

capchars (function)

Use capchars to find out the number of characters in the capture file.

Format

```
x = capchars
```

capchars checks the number of characters currently in the capture file and returns an integer.

For related information, see the capfile function and the capture statement.

Example

```
if capchars >= 10000 then capture off
```

In this example, capture is turned off if there are more than 10,000 characters in the capture file.

capfile (function)

Use `capfile` to find out the name of the current capture file, if one is open.

Format

```
x$ = capfile
```

The `capfile` function returns the name of the current capture file. A null string is returned if `capture` is set to `off`.

For related information, see the `capture` statement.

Example

```
print capfile
```

In this example, the name and path of the capture file are printed on the screen.

capture (statement)

Use capture to control the capture of incoming data.

Format

```
capture [{new | to}] <filename>

capture {on | pause | toggle | / | off}
```

The capture statement controls whether data capture is active at any particular time. The capture facility is available to collect data coming in from the communications port. Data is captured in the directory specified for capture files. This directory can be specified by setting the dirfil and downloaddir system variables (see dirfil and downloaddir later in this chapter).

The capture options are described in Table 6-3.

Table 6-3. Capture options

Option	Description
new	Turns capture on, and specifies the name of a file in which to capture the incoming data. If the file already exists, it is deleted before the new data is added to the file. If backups is on, the old file is renamed to .BAK, thus preserving the contents of that file. The capfile function returns the file name. If you use capture new without an argument, an error occurs.
to	Turns capture on, and specifies the name of the file in which to capture incoming data. If the file already exists, the newly captured data is appended to the end of the file. You can check the specified file name with the capfile function.

continued

Table 6-3. Capture options (cont.)

Option	Description
`on`	Turns `capture` on if it was off. If `capture` is turned on after being off, CASL synthesizes a capture file name using the `name` session setting and the current date (the month is a single digit: valid digits are 1–9 for January to September and A–C for October to December). For example, a file captured from the MCIMAIL session on January 1 is MCI.101; an entry captured on December 21 is MCI.C21.
`pause`	Suspends data capture. Data already captured is retained in the buffer. You can restart `capture` with the `capture on` or `capture toggle` commands, or terminate it with the `capture off` command.
`toggle`	Causes `capture` to toggle `on` if it was in `off` or `pause` state; if `capture` was `on`, `toggle` changes the state to `pause`.
`/`	This is an alternative to the `toggle` option. If you need to toggle capture often, assign the following script to a function key: `capture /`
`off`	Stops data capture and closes the file.

Versions of Crosstalk for Windows older than 2.0 do not support the `to` option.

Note: You can control `capture` using your Crosstalk application in the following ways:

- Choose Session from the Action pull-down and then choose Start Capture.
- Choose the Capture icon from the QuickBar. ■

For related information, see the `capfile` and `capchars` functions and the `grab` statement.

Examples

```
capture on
```

In this example, the script will begin capturing data.

```
capture new "vutext.doc"
```

In this example, data is captured in a new file called `VUTEXT.DOC`. Any previous file named "VUTEXT.DOC" in that directory is deleted, unless `backups` is `on`.

case ... endcase (statements)

Use `case ... endcase` to perform statements based on the value of a specified expression.

Format

```
case <expression> of
   <list of values> : <statement group>
   <list of values> : <statement group>
   ...
   ...
   [default : <statement group>]
endcase
```

`case` lets you take a variety of actions based on the value of a particular expression. *`expression`* can be any type of expression or variable. *`list of values`* is a list of expected values for *`expression`* and must match the data type of *`expression`*. The values can be constants or expressions and must be separated by commas if you use more than one value on a logical line.

`statement group` is a series of statements to perform if one of the items in *`list of values`* matches the current expression. After the associated *`statement group`* has been performed, the script continues to execute at the point after the `endcase` statement (unless, of course, control was transferred somewhere else with a `goto` or a `gosub` statement).

`default` and its associated *`statement group`* describe a statement or group of statements to perform if none of the other values match. If you include `default`, be sure it is the last item in the list. `endcase` denotes the end of the `case/endcase` construct.

You can nest `case ... endcase` statements.

Versions of Crosstalk for Windows older than 2.0 do not support these statements.

For related information, see the `gosub`, `goto`, `if ... then ... else`, and `watch ... endwatch` statements.

Examples

```
label ask_again
print "Please choose a number (0-4): " ;
input choice
print
case choice of
   0, 4     : end
   1        : goto choose_speed
   2        : goto main_menu
   3        : goto save_setup
   default  : goto ask_again
endcase
```

In this example, `case` examines the value of the integer variable `choice`. If `choice` is 0 (zero) or 4, the script ends. If `choice` has a value between 1 and 3, the script branches to the appropriate label. If `choice` is not 0 (zero) through 4, the `default` action is taken. If none of the conditions were met (assuming a default was not provided), the script would continue execution at the statement following the `endcase`.

```
case left(date, 5) of
   "08/12" : print "Today is Aaron's birthday!"
   "07/04" : print "Why are you here today?"
   "10/31" : alarm 6 : print "Boo!"
endcase
```

This example shows that you can use `case` with any type of expression. The actions taken in this example depend on the date.

chain (statement)

Use chain to compile and run a script.

Format

```
chain <filename> [, <args>]
```

args represents an optional argument list that contains the individual arguments to be passed to the other script. Individual arguments must be separated by commas.

chain compiles and runs a script source (.xws) file if there is no compiled version of the script, or if the date of the source file is more current than the date of the compiled version. Otherwise, chain runs the compiled version of the script. Script names do not require an extension.

Note: The script that issues a chain statement ends and is removed from memory; therefore, control cannot be passed back to it. ■

Versions of Crosstalk for Windows older than 2.0 allow a label to be supplied in parentheses. This is no longer allowed.

For related information, see the arg function and the do statement.

Example

```
chain "menu", "arg1", "arg2"
```

In this example, the script chains to a script called MENU and passes the script 2 arguments.

chdir (statement)

Use `chdir` to change the current disk directory.

Format

```
chdir <string>
```

string must be an expression containing a valid directory name. The current working directory is set to the new value. This does not change the current drive designation.

Versions of Crosstalk for Windows older than 2.0 reset the current directory when the script ends. The new directory is now preserved.

Note: You can also use the abbreviation `cd` for this statement. ■

For related information, see the `drive` statement.

Examples

```
chdir "C:\XTALK"
```

In this example, the directory is changed to `XTALK`.

```
chdir dirname
```

In this example, the directory is changed to the directory name stored in the script's `dirname` variable.

Win chmod (statement)

Use chmod to change the attributes of a file.

Format

chmod <*filename*> [, <*attribute*>]

filename must be a string expression containing a valid file name, which may contain drive and path specifiers.

attribute is optional. If it is specified, it must be an integer expression containing a valid file attribute. If attribute is not specified, the file is set to "normal" attributes.

The attribute is specified as a bitmap, with the bits having the values shown in Table 6-4. As with any bitmap, values are added together for multiple conditions.

Table 6-4. Bitmap values for the chmod statement

Hex	Dec	Attribute/Meaning
01h	1	A read-only file.
02h	2	A hidden file. The file is excluded from directory searches.
04h	4	A system file. The file is excluded from directory searches.
08h	8	The volume name of the disk.
10h	16	A subdirectory.
20h	32	An archive bit. This bit is set by DOS whenever a file has been written to and closed. It indicates the file has been changed since it was last backed up.
40h	64	Undefined and reserved by DOS.
80h	128	Undefined and reserved by DOS.

Caution: Be very careful when you use `chmod`; you can cause files to disappear from your directory list if they are hidden. ■

Examples

```
chmod "XTALK.EXE", 1
```

In this example, the file, `XTALK.EXE`, becomes read-only.

```
chmod "secret.fil", 3
```

In this example, the file, `secret.fil`, becomes read-only and hidden. ■

choice (system variable)

Use choice to check the value of the pushbutton that dismissed a dialog box.

Format

```
n = choice
```

choice contains the value identifying the pushbutton used to exit a dialog box.

Examples

```
dialogbox 20, 50, 280, 100
   defpushbutton 10, 10, 80, 80, "Choice 1", ok
   pushbutton 100, 10, 80, 80, "Choice 2", cancel
   pushbutton 190, 10, 80, 80, "Choice 3", ok
enddialog
print "Choice was "; choice
```

In this example, choice has a value of 1 if the Choice 1 (ok) pushbutton is chosen, 2 if the Choice 2 (cancel) pushbutton is selected, or 3 if the Choice 3 (ok) pushbutton is chosen.

```
dialogbox 20, 50, 280, 100
   pushbutton 100, 10, 80, 80, "Choice 1", cancel
   pushbutton 190, 10, 80, 80, "Choice 2", ok
   defpushbutton 10, 10, 80, 80, "Choice 3", ok
enddialog
print "Choice was "; choice
```

In this example, choice has a value of 1 if the Choice 1 (cancel) pushbutton is chosen, 2 if the Choice 2 (ok) pushbutton is selected, or 3 if the Choice 3 (ok) pushbutton is chosen. Note that in both of these examples, the pushbuttons are displayed in the same locations in the dialog box.

chr (function)

Use chr to get a single character string defined by an ASCII value.

Format

```
x$ = chr(<integer>)
```

chr returns a 1-byte string that contains the character with the ASCII value contained in *integer*.

integer is a decimal number that is converted to its Modulo 255 value; therefore, it is in the range of 0–255.

Examples

```
cr = chr(13)
```

In this example, the variable cr is set to ASCII value 13, which is a carriage return.

```
reply chr(3)
```

In this example, the script sends ASCII value 3 to the host.

cksum (function)

Use cksum to get an integer checksum for a string of characters.

Format

```
x = cksum(<string>)
```

cksum returns the arithmetic checksum of the characters contained in *string*. *string* can be any length. You can use this function to develop a proprietary file transfer protocol, or to check the integrity of a string transferred between two systems using a non-protocol transfer.

For related information, see the crc function.

Examples

```
check = cksum(what_we_got)
```

In this example, the checksum value of the what_we_got variable is stored in the check variable.

```
if cksum(data_in) <> cksum(data_out) then alarm
```

In this example, the script sounds an alarm if the checksum of the data_in variable is not the same as the checksum of the data_out variable.

class (function)

Use `class` to get the Crosstalk class value for a single-character string.

Format

```
x = class(<string>)
```

`class` returns the "class number" bitmap of the first character in *string*.

The bitmap value returned indicates the class(es) in which the first character in the string falls. Classes define such groupings as capital letters (A–Z), decimal digits (0–9), and hexadecimal digits (0–9 plus A–F or a–f). Table 6-5 lists class groupings.

Table 6-5. Class groupings

Hex	Dec	Class contents
01h	1	White space (space, tab, CR, lf, ff, bs, null)
02h	2	Uppercase alpha (A–Z)
04h	4	Lowercase alpha (a–z)
08h	8	Legal identifier ($, %, _)
10h	16	Decimal digit (0–9)
20h	32	Hexadecimal digit (A–F, a–f)
40h	64	Delimiters: space, comma, period, tab, (, /, \, :, ;, <, =, >, !
80h	128	Punctuation: !-\, :-@, [-^, {-~

A character may fall into more than one class: the comma, for example, is both a delimiter and a punctuation mark, and returns a `class` value of 0C0h or 192 decimal.

Example

```
x = class(a_char) : if x = 1 then ...
```

In this example, `a_char` is a white space if `x` is 1.

clear (statement)

Use `clear` to clear the terminal screen.

Format

```
clear [window] [, line] [, eow] [, bow] ...
   [, eol] [, bol]
```

If no option is specified, the entire window is cleared and the cursor moves to the top left corner of the window. If an option is specified, the cursor remains in place. Table 6-6 explains the options.

Table 6-6. Options for the clear statement

Option	Explanation
`window`	Clears the entire window.
`line`	Clears the line on which the cursor is located.
`eow`	Clears from the cursor to the end of the window.
`bow`	Clears from the cursor to the beginning of the window.
`eol`	Clears from the cursor to the end of the current line.
`bol`	Clears from the cursor to the beginning of the current line.

Examples

```
clear bow
```

In this example, the script clears the session window from the cursor back to the beginning of the window.

```
clear window
```

In this example, the script clears the entire session window.

close (statement)

Use `close` to close an open data file.

Format

`close [# <filenum>]`

`close` ends access to an open file. If *filenum* is not given, all open files are closed. Note that all open files are closed when the script that opened them terminates.

The # symbol must precede the file number.

For related information, see the `open` statement.

Example

```
close
```

In this example, all open files are closed.

cls (statement)

The `cls` statement, which is a synonym for the `clear` statement, is supported only for backward compatibility. Refer to `clear` earlier in this chapter.

cmode (system variable)

Use cmode to control the capture mode.

Format

```
cmode = {"normal" | "raw" | "visual"}
```

The capture buffer is available to collect data coming in from the communications port. The cmode system variable controls the appearance of the captured data through its options, which are outlined in Table 6-7.

Table 6-7. Options for the cmode variable

Option	Description
normal	The data is captured in the order received, but with terminal control sequences removed, producing generally readable text that can be used by other programs or scripts. In this mode, the backspace character erases the last character captured, and CR and LF characters are paired appropriately.
raw	All data is captured as received, without removal of terminal control characters.
visual	Data is captured as it looks on the screen; however, due to terminal control sequences, it may be in a different order than the one in which it was received. Data is passed to the buffer when the screen is cleared or when lines are scrolled off the screen. Data that is selectively erased by the host cannot be captured.

Example

```
cmode = "raw"
```

In this example, cmode is set to "raw". All data will be captured as received, without removing terminal control characters.

compile (statement)

Use `compile` to compile a script file.

Format

`compile <filename>`

This statement causes the specified script to be compiled. The compiled script file is saved in the same directory where the source script is found.

Example

```
compile "MENU"
```

In this example, the script tells the compiler to compile a script called MENU.

connected (function)

The `connected` function, which is a synonym for the `online` function, is supported only for backward compatibility. Refer to `online` later in this chapter.

connectreliable (module variable)

Use `connectreliable` to determine if there is a reliable, or error-free, connection.

Format

```
x = connectreliable
```

`connectreliable` is `true` if the modem connection is reliable, `false` if it is not.

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support this variable.

For related information, see the `assume` statement.

Example

```
assume device "DCAMODEM"
if connectreliable then
{
   assume protocol "DCAXYMDM"
   protocol = "DCAXYMDM"
   protomodel = "YMODEM/G"
}
```

In this example, the script tells the compiler to assume the module type `device` with the name `DCAMODEM`. If this device provides an error-free connection, the script assumes the module type `protocol` with the name `DCAXYMDM` and then sets two variables to the appropriate values.

copy (statement)

Use `copy` to copy a file or group of files.

Format

`copy [some] <filespecfrom>, <filespecto>)`

`filespecfrom` must be a legal file name (full path names and wild cards are permitted). *`filespecto`* specifies the new drive-path-file name for the copy of the file and defaults to the current directory.

If you specify `some`, the user must approve each file before it is copied.

Versions of Crosstalk for Windows older than 2.0 do not support the `copy` statement.

Examples

```
copy "menu.xts", "menu2.xts"
```

In this example, `menu.xts` is copied to `menu1.xts`.

```
copy "*.xts", "*.bak"
```

In this example, the script makes a copy of each file with the `.xts` extension and gives the copied files a `.bak` extension.

```
copy some "*.xts", "A:"
```

In this example, the script copies all files with the `.xts` extension to drive A, but confirmation is requested of the user before each individual file is copied.

count (function)

Use count to determine the number of occurrences of a character within a string.

Format

```
x = count(<string1>, <string2>)
```

count returns the number of times any of the characters in *string2* occur in *string1*. This function can take the place of the instr function in a counting loop to determine how many times your script must take some future action.

This function is case-sensitive.

For related information, see the instr function.

Examples

```
x = count("sassafras", "s")
```

In this example, count returns the number of times the letter "s" occurs in the string. The result is 4.

```
x = count("sassafras", "sa")
```

In this example, count returns the number of times the letters "s" and "a" occur in the string. The result is 7.

crc (function)

Use crc to determine the cyclical redundancy check value for a string.

Format

```
x = crc(<string> [, <integer>])
```

x is returned as the crc of *string*. The crc starts with a value of 0 (zero) unless a starting value is given in *integer*.

As with the cksum function, you can use crc to develop a proprietary file transfer protocol or to check the integrity of a string.

For related information, see the cksum function.

Examples

```
x = crc("Crosstalk")
```

In this example, x is assigned the crc value of the string Crosstalk.

```
x = crc(text_line)
```

In this example, x is assigned the crc value of the text_line variable.

curday (function)

Use curday to find out the current day of the month.

Format

```
x = curday
```

curday returns the current day of the month. The returned value is always in the range of 1–31.

Examples

```
x = curday
```

In this example, x is set to the current day of the month.

```
if curday = 15 then gosub pay_bills
```

In this example, control passes to the subroutine pay_bills if the current day is day 15.

curdir (function)

Use curdir to check the name of the current directory.

Format

```
x$ = curdir[(<string>)]
```

curdir returns the current directory of the drive specified by *string*. If you do not specify *string*, curdir returns the directory of the current drive. curdir returns a null string if the specified drive is not available.

For related information, see the curdrive function.

Examples

```
where_we_are = curdir
```

In this example, curdir stores the name of the current directory in the where_we_are variable.

```
whats_on_a = curdir("a:")
```

In this example, curdir stores the name of the current directory for drive A: in the whats_on_a variable.

Win curdrive (function)

Use `curdrive` to find out the current default drive.

Format

```
x$ = curdrive
```

`curdrive` returns a 2-character string consisting of the letter of the current default drive followed by a colon.

For related information, see the `curdir` function.

Examples

```
what_we_are_on = curdrive
```

In this example, `curdrive` stores the letter of the current drive in the `what_we_are_on` variable.

```
if curdrive > "C:" then ...
```

In this example, the script takes some action if the letter of the current drive is greater than C (D, E, F, and so on). ■

curhour (function)

Use curhour to get the current hour in a 24-hour format.

Format

```
x = curhour
```

curhour returns an integer value containing the current hour, in the range of 0–23.

Examples

```
x = curhour
```

In this example, curhour sets the variable x to the number for the current hour.

```
if curhour = 23 then chain "CALLBBS"
```

In this example, the script chains to a script called CALLBBS if curhour is set to 23.

curminute (function)

Use `curminute` to get the current minute.

Format

```
x = curminute
```

`curminute` returns an integer containing the current minute, in the range of 0–59.

Examples

```
x = curminute
```

In this example, `x` is set to the current minute.

```
if curminute = 30 then ...
```

In this example, the script tests whether the current minute is equal to 30.

curmonth (function)

Use curmonth to get the number of the current month.

Format

```
x = curmonth
```

curmonth returns an integer value containing the current month, in the range of 1–12.

Examples

```
x = curmonth
```

In this example, x is set to the current month.

```
if curmonth = 12 then capture "DECEMBER.DAT"
```

In this example, the script captures data in the DECEMBER.DAT file if the current month is 12.

cursecond (function)

Use `cursecond` to get the current second.

Format

```
x = cursecond
```

`cursecond` returns an integer value containing the current second, in the range of 0–59.

Examples

```
x = cursecond
```

In this example, `x` is set to the current second.

```
if cursecond = 30 then ...
```

In this example, the script tests whether the current second is equal to 30.

curyear (function)

Use curyear to find out the current year.

Format

```
x = curyear
```

curyear returns an integer value containing the current year.

Examples

```
x = curyear
```

In this example, x is set to the current year.

```
if curyear = 1992 then capture "DEC1992.DAT"
```

In this example, data is captured in the DEC1992.DAT file if the current year is 1992.

cwait (statement)

Use cwait to control ASCII text uploading by pacing individual characters.

Format

```
cwait {none | echo | delay <integer>}
```

cwait (character wait) controls text uploads by defining the condition to be met before a character can be sent to the host computer. The options for cwait are explained in Table 6-8.

Table 6-8. Options for the cwait statement

Option	Explanation
none	Do not wait after each character. Send each character as fast as possible. This allows the fastest uploads.
echo	Wait until the host sends back the character just transmitted, then send the next character. This method is slow, but it is the best choice when sending files to host systems that cannot accept data at full speed.
delay	Wait *integer* milliseconds before sending the next character. Use this when the host does not echo the characters uploaded but cannot accept text at full speed. The maximum number that can be entered is 9999 (9.999 seconds). Note that in GUI environments, the delay time may actually be greater than the value specified.

You can use cwait in conjunction with the lwait statement to control the speed of text uploads to host computers. Many computers expect to receive input at about 80 words per minute (wpm) from a human typist, not at the 3,000 wpm (at 2,400 bps) speed that text is uploaded from a computer.

Only one `cwait` setting can be in effect at any one time.

Use `cwait` only when you are on line; however, you can set the parameters while on line or off line.

For related information, see the `lwait` and `wait` statements.

Examples

```
cwait echo
```

In this example, the script waits for transmitted characters to be echoed by the host.

```
cwait delay 3
```

In this example, the script waits at least 3 milliseconds (.003 seconds) between each character.

date (function)

Use `date` to return a date string.

Format

```
x$ = date[(<integer>)]
```

This function works two ways. First, if *integer* is not specified or has a zero value, `date` returns a string containing the current system date. The returned string is in the format appropriate for the country where the computer is operating, for example, mm/dd/yy for the U.S.A. and dd/mm/yy for most European countries.

In the second way, *integer* specifies the number of days elapsed since January 1, 1900. `date` returns the date string for that day. This second option is most useful for converting the results of the `filedate` function to a "normal" string.

Note: If you want to check for a specific date, use the `curday`, `curmonth`, and `curyear` functions. ■

For related information, see the `filedate` , `curday`, `curmonth`, and `curyear` functions.

Examples

```
x = date(31354)
```

In this example, the script sets `x` to "11/04/85".

```
if right(date(filedate("XTALK.EXE")), 2) > "87" then
```

This seemingly complex line is actually doing something fairly simple. First, it gets the file date of the XTALK.EXE file using the `filedate` function, converts that to standard date format using the `date` function, and then uses the `right` function to get the 2 rightmost characters. If those 2 characters are a number greater than 87, some action is taken.

definput (system variable)

Use definput to select a default file number for input.

Format

```
definput = <filenum>
```

filenum must be an integer expression. definput lets you specify a default file number for all file input operations that follow the definput declaration. seek, get, read, and read line assume the file number specified by definput if no explicit file number is provided.

The combination of the freefile function and the definput variable can produce file manipulation modules that can make subsequent coding easier and more flexible.

This variable is valid only for files opened in input or random mode.

For related information, see the freefile function and the get, open, read, read line, and seek statements.

Example

```
fileno = freefile
open input "f.dat" as #fileno
definput = fileno
```

This example uses the freefile function to get the next free file number, opens a file with the open statement, and then assigns the file number to the definput system variable. Subsequent file operations (for example, read) for this file need not specify the file number.

defoutput (system variable)

Use defoutput to select a default file number for output.

Format

```
defoutput = <filenum>
```

filenum must be an integer expression. defoutput lets you specify a default file number for all file output operations that follow the defoutput declaration. put, write, and write line assume the file number specified by defoutput if no explicit file number is provided.

This variable is valid only for files opened in output or random mode.

For related information, see the open, seek, put, write, and write line statements.

Example

```
fileno = freefile
open output "f.dat" as #fileno
defoutput = fileno
```

This example uses the freefile function to get the next free file number, opens a file with the open statement, and then assigns the file number to the defoutput system variable. Subsequent output operations (for example, write) for this file need not specify the file number.

dehex (function)

Use dehex to convert an enhex string back to its original format.

Format

```
x$ = dehex(<string>)
```

dehex converts a string of ASCII characters in hexadecimal format back to a string of binary data.

Since each byte in *string* is a 2-byte hexadecimal representation, the string returned by dehex is half as long as *string*.

Like entext and detext, enhex and dehex are complementary functions designed to permit the exchange of binary information over communications services that allow only 7-bit transfers; many of the electronic mail systems allow the transfer of only 7-bit ASCII information.

Binary data strings that have been converted with enhex require dehex to restore the 8-bit binary format.

For related information, see the detext, enhex, and entext functions.

Examples

```
program_line = dehex(sendable)
```

In this example, dehex converts the ASCII hexadecimal string sendable to binary and returns the result in program_line.

```
spread_sheet_line = dehex(nextline)
```

In this example, dehex returns the binary equivalent of nextline in spread_sheet_line.

delete (statement)

Use the `delete` statement to delete files from the disk.

Format

`delete [noask] <filespec>`

`delete` removes a file from the disk. *`filespec`* must be a valid file specification, which can contain drive and path specifiers. If *`filespec`* contains wild cards, the user is asked to confirm each file fitting the file specification.

Use `noask` to suppress user intervention.

Examples

```
delete "script1.xws"
```

In this example, the file `script1.xws` is deleted.

```
input f$ : delete f$
```

In this example, the script accepts the file name entered by the user and then deletes the file.

delete (function)

Use the delete function to remove characters from a string.

Format

```
x$ = delete(<string> [, <start> [, <length>]])
```

delete returns *string* with *length* characters removed beginning at the character represented by *start*. If *length* is not specified, one character is removed. If *start* is omitted, the deletion starts at the first character position in *string*.

start must be in the range 1 <= *start* <= length(*string*).

If *start* + *length* is greater than length(*string*), the leftmost *start* -1 bytes are returned.

Example

```
dog_name = delete("Fixxxdo", 3, 3)
```

In this example, the script deletes 3 characters, starting at position 3, from the string Fixxxdo. The result is "Fido."

description (system variable)

Use `description` to read or set the description of the current session.

Format

`description = <string>`

`description` sets and reads the descriptive text associated with the current session. Only 40 characters are displayed. You can set the description to a null string (" ").

For related information, see the `name` function.

Example

```
description = "Crosstalk Communications BBS"
```

In this example, the script sets `description` to the indicated string.

destore (function)

Use destore to restore strings converted with the enstore function back to their original form.

Format

```
x$ = destore(<string>)
```

destore converts strings of printable ASCII characters,which have been converted with enstore, back to their original, embedded control character form.

Control characters in caret notation such as ^G, are converted back to control characters, in this case a Ctrl-G (bell) character. The vertical bar (|) is translated to a Ctrl-M (CR).

destore does not convert a caret preceded by a backquote character (`); however, the backquote character is discarded since it is no longer needed for protection. Therefore, `^G becomes ^G.

You must have created *string* with enstore.

For related information, see the enstore function.

Example

```
line_to_show_user = destore(password)
```

In this example, destore converts the string password back to its original form and returns the result in line_to_show_user.

detext (function)

Use detext to convert an entext string back to its original form.

Format

```
x$ = detext(<string>)
```

This function works in tandem with the entext function to provide a method of transferring 8-bit data over 7-bit networks. entext takes binary data and converts it to normal 7-bit ASCII characters (the result may even be readable); detext takes the entext data and converts it back to its original form.

You must have originally converted *string* with entext.

For related information, see the entext function.

Example

```
convtd_text = detext(ntxtd_string)
```

In this example, detext converts ntxtd_string from 7-bit ASCII characters to 8-bit binary form and returns the result in convtd_text.

device (system variable)

Use device to read or set the connection device for the current session.

Format

device = <*string*>

The device variable specifies the communications device for the current session. Table 6-9 lists the applicable devices.

Table 6-9. Connection devices

Device name	Sub-models (use the devmodel variable)	Functionality
DCASERIL* or Serial Tool†	(None)	Loads the serial connection tool.
DCAMODEM* or Apple Modem Tool†	(None)	Loads the modem connection tool.
DCANASI*	(None)	Loads the Novell® NASI connection tool.
DCAINT14*	(None)	Loads the INT 14 connection tool.

* Windows environment
† Macintosh environment

After setting this variable, use the assume statement to gain access to the device variables.

Note: To set the equivalent parameter using your Crosstalk application, choose Connection from the Settings pull-down. ■

Versions of Crosstalk for Windows older than 2.0 do not support this variable.

For related information, see the `assume` statement and the `protocol` and `terminal` system variables.

Example

```
assume device "DCAMODEM"
device = "DCAMODEM"
port = 1
```

This example shows how to load the modem connection tool and set the communications port to COM1.

dialmodifier (module variable)

Use `dialmodifier` to set the dialing modifier string.

Format

`dialmodifier = <string>`

`dialmodifier` changes the way Crosstalk dials for each session. The maximum length of this variable is 16 characters.

You can use this variable only with Hayes® or Hayes command-compatible modems (those that use the "AT" command set).

For versions of Crosstalk for Windows older than 2.0, this variable was called `modifier`.

Example

```
dialmodifier = "M0"
```

In the example, `dialmodifier` is set to "M0". Crosstalk inserts the dialing modifier in the dialing prefix. If the dialing prefix is "ATDT", when the modem is dialed, the modem sends out "ATM0DT".

dialogbox ... enddialog (statements)

Use `dialogbox ... enddialog` to create custom dialog boxes.

Format

```
dialogbox <x,y,w,h> [, caption]
   [<defpushbutton x, y, w, h, string [, options]>]
   [<pushbutton x, y, w, h, string [, options]>]
   [<ltext x, y, w, h, string>]
   [<ctext x, y, w, h, string>]
   [<rtext x, y, w, h, string>]
   [<edittext x, y, w, h, init_text, str_result_var ...
      [, options]>]
   [<radiobutton x, y, w, h, string, result_var ...
      [, options]>]
   [<checkbox x, y, w, h, text_str, result_var ...
      [, options]>]
   [<groupbox x, y, w, h, title>]
   [<listbox x, y, w, h, comma_string, ...
      int_result_var [, options]>]
   [<listbox x, y, w, h, string_array, ...
      int_result_var [, options]>]
enddialog
```

This statement is useful for designing a user interface for your scripts. Using the `dialogbox/enddialog` construct, you can create dialog boxes that are easy to use and work like standard dialog boxes.

All variables used in a dialog box must be defined before the `dialogbox/enddialog` construct. The values assigned to variables for `radiobutton`, `checkbox`, and `listbox` are used to set the initial value of these dialog items. For `radiobutton` and `checkbox`, setting the boolean variable *`result_var`* to `true` selects it, `false` does not. For `listbox`, setting the integer variable *`int_result_var`* determines which item in the list box is highlighted. The range is limited by the number of items in the list. You can use `caption` to define a title for the dialog box.

You can examine the variables after the `dialogbox/enddialog` construct to determine the choices made by the user. The system variable `choice` contains the value that corresponds to the pushbutton used to exit the dialog box. For example, if the first pushbutton is chosen, `choice` is set to 1 (one). Note that no variables are updated if the Cancel pushbutton is used.

Unless otherwise specified, Crosstalk defines the first letter of a pushbutton or prompt-text string as an accelerator. Placing an ampersand (&) in a string used for the text allows you to define your own accelerator. The letter after the ampersand becomes the accelerator.

Dialog items

`defpushbutton`, `ltext`, `ctext`, `rtext`, `edittext`, `radiobutton`, `pushbutton`, `checkbox`, `groupbox`, and `listbox` are known as dialog items.

x and *y* for `dialogbox` are the pixel coordinates for the window. *w* and *h* are the width and height of the dialog box.

The *x*, *y*, *w*, and *h* for dialog items are the same, but work within the dialog box created with the `dialogbox/enddialog` construct. A horizontal unit is 1/4 of a system font character; a vertical unit is 1/8 of a system character font. The origin of *x* and *y* is 0,0, which is the top left corner of the dialog box.

`defpushbutton` is a special type of pushbutton. It is the default pushbutton, so it has a bold border. You would normally use `defpushbutton` to display the dialog's OK pushbutton. In essence, this pushbutton is "pushed" when the user presses ENTER. See `pushbutton` for more information.

`pushbutton` displays a choice a user can make to exit a dialog box, such as OK, CANCEL, SETTINGS, and so on. Any dialog box must have at least one pushbutton. If there is only one, use the `defpushbutton` dialog item. When the user exits the dialog box, the variable `choice` is assigned the number of the pushbutton used to exit the dialog box. For instance, if the second pushbutton is chosen, `choice` is set to 2, or if the fourth pushbutton is selected, `choice` is set to 4. The script can then check `choice` to take appropriate action.

The width should be the length of `(`*`string`*` * 4) + 10`. The height is usually 14.

`ltext` (left text), `rtext` (right text), and `ctext` (center text) display text and define its justification in the dialog box. The width should be 4 times the length of *`string`*. The height is usually 8.

`edittext` displays an edit box for user input. The string entered in the edit box is returned in *`str_result_var`*. Precede `edittext` with `ltext`, `rtext`, or `ctext` to display a prompt for the edit box. The width of the text box should be at least 4 times the maximum length of the string the user may enter. The height is usually 12.

`radiobutton` displays a round radio or option button that is chosen when clicked. Radio buttons are usually found in groups of several, horizontally placed in a dialog box. The first `radiobutton` in a group must have the `tabstop group` option set, or the arrow keys may not work properly in the dialog box. The first dialog item used after a group of `radiobutton` definitions must also have the `tabstop group` option, so that the operating environment knows where one group ends and the next one begins. *`result_var`* is `true` if the radio button is selected, `false` if not. You must examine *`result_var`* for each `radiobutton` defined until you find one that is set to `true`.

The width of a `radiobutton` is generally the length of `(`*`string`*` * 4) + 10`. The height is generally 10.

`checkbox` displays a square box, which is checked or unchecked as the user clicks on the item. After the user exits the dialog box, *`result_var`* is true or false depending on whether the check box was checked or not.

The width of a `checkbox` should be at least the length of `(`*`text_str`*` * 4) + 10`. The height is usually 12.

`groupbox` draws a box for a group of dialog items yet to be defined. The title string appears in the upper border of the box. Dialog item definitions for this box should follow.

`listbox` displays a list box containing the comma-delimited strings in *`comma_string`*. The number of the list box item chosen is returned in *`int_result_var`*. Zero is returned if no item was chosen.

The width of a list box should be at least 4 times the length of the longest string in *`comma_string`*. The height should be 8 times the number of items from *`comma_string`* that you want to display at one time. The height of the list box is limited by the height of the dialog box.

If an array of strings (*`string_array`*) is specified for `listbox` instead of a *`comma_string`*, an array is displayed. Note that the array must be single dimensional with an alternative lower boundary of 1 (one).

The width of a list box should be at least 4 times the length of the longest string in *`string_array`*. The height should be 8 times the number of items from *`string_array`* that you want to display at one time.

Dialog item options

`tabstop`, `tabstop group`, `focus`, `ok`, and `cancel` are options for some of the dialog items, which include `defpushbutton`, `pushbutton`, `edittext`, `radiobutton`, `checkbox`, and `listbox`.

`tabstop` defines the dialog items to which you can tab if the user is using the keyboard rather than the mouse.

`tabstop group` marks the beginning or end of a group of radio buttons. Radio buttons are generally a group of horizontally placed buttons. Use the TAB key to get to the first button in the group, then use the arrow keys to move from one button to the next. Pressing TAB again takes you to the next group (the next dialog item outside the radio button group).

`focus` defines where to place the focus (cursor) for the dialog box. If this is not used, the focus is set at the first `tabstop` in a dialog box.

`ok` is for a `pushbutton` only. This identifies the pushbutton to associate with the ENTER key. In general, you use this option only with `defpushbutton`.

`cancel` is for a `pushbutton` only. This identifies the pushbutton to associate with the ESC key.

Note: This statement supports dialog box comments and flow control of the logic related to displaying a dialog box. Versions of Crosstalk for Windows older than 2.0 do not support these features. Crosstalk Mark 4 does not support this statement. ■

For related information, see the `alert` statement.

Examples

```
dialogbox 61, 20, 196, 76
   ltext    6, 4, 148, 8, 'About calling CompuServe ' + ...
   'directly ...'
   ltext    6, 24, 176, 8, 'When setting up Crosstalk ' + ...
   'to call CompuServe'
   ltext    6, 36, 188, 8, 'Directly, you must leave ' + ...
   'the NetID field blank.'
   defpushbutton 80, 56, 36, 14, 'Ok', tabstop
enddialog
```

This example displays a simple dialog box that provides some information for the user. The user can read the text and choose OK when ready to continue.

```
/*
Dialog box example
*/
string  edit$
boolean check1, check2, check3,
boolean radio1, radio2
integer list1
string items[1:8]

label SampleDialog

check1 = true          -- true shows the check box selected
check2 = true
check3 = true
list1  = 3             -- a 3 will highlight the 3rd item in
                       -- the list
radio1 = true          -- true will show the radio button
                       -- selected
```

```
radio2 = false        -- false shows that the radio button is
                      -- not selected
items[1] = "Item1"    -- array elements 1 through 8
items[2] = "Item2"
items[3] = "Item3"
items[4] = "Item4"
items[5] = "Item5"
items[6] = "Item6"
items[7] = "Item7"
items[8] = "Item8"

dialogbox 34, 23, 253, 125
   ltext 4, 4, 86, 8, "Dynamic Dialog"
   groupbox 4, 18, 197, 52, "Crosstalk for Windows"
   checkbox 12, 30, 154, 12, "Designed for the " + ...
      "Windows environment", check1, tabstop
   checkbox 12, 42, 150, 12, "Includes a powerful " + ...
      "script language", check2, tabstop focus
   checkbox 12, 54, 170, 12, "Full Dynamic Data " + ...
      "Exchange (DDE) support", check3, tabstop
   listbox 4, 74, 72, 40, items, list1, tabstop
   ltext 87, 76, 44, 8, "Enter text:"
   edittext 135, 76, 94, 12, "", edit$, tabstop
   radiobutton 88, 91, 93, 12, "Radio Button 1", ...
      radio1, tabstop group
   radiobutton 88, 103, 93, 12, "Radio Button 2", radio2
   defpushbutton 208, 22, 36, 14, "Ok", ok tabstop group
   pushbutton 208, 39, 36, 14, "Cancel", cancel ...
      tabstop
enddialog
```

This example produces a more complex dialog box that contains check boxes, a list box, edit boxes, and radio buttons.

dirfil (system variable)

Use `dirfil` to read or set the directory used for transfers and captures.

Format

`dirfil = <string>`

`dirfil` checks or sets the directory used for file transfers and data capture.

The Crosstalk installation program creates `dirfil` for transfers and captures. The default path for Windows users consists of the directory where the XTALK.INI file is located and the Crosstalk FIL directory. For example, if XTALK.INI is in the \XTALK directory, the `dirfil` setting is \XTALK\FIL. The default path for Macintosh users consists of the Download Files folder in the folder where the Crosstalk application is located.

This variable is not supported for Crosstalk Mark 4.

For related information, see the `downloaddir` system variable.

Examples

```
dirfil = "c:\xtalk\fil"
```

In this example, `dirfil` is set to C:\XTALK\FIL directory.

```
if exists(dirfil+("\TEST.DAT")) then ...
```

In this example, the script tests whether the file TEST.DAT exists in the `dirfil` directory.

display (system variable)

Use display to enable or disable the display of incoming characters.

Format

```
display = {on | off}
```

display controls the display of incoming characters. If display is off, then incoming information is not displayed.

Characters sent to the screen with the print statement are considered incoming characters, and are not displayed if display is off.

display is active only while the script that is using it is running.

For related information, see the print statement.

Example

```
wait for "Password:"
display = off
reply password
display = on
```

In this example, the script waits for the "Password:" prompt from the host. When the prompt is received, display is turned off, the contents of the system variable password are sent to the host, and display is turned back on.

do (statement)

Use do to compile and run a script.

Format

do <*filename*> [, <*args*>]

The do statement, like the chain statement, invokes another script and passes control to that script. Unlike the script that uses the chain statement, however, the script issuing the do statement does not terminate after it invokes the "child" script; rather, it waits until the other script returns control.

args represents an optional argument list that contains the individual arguments to be passed to the other script. Individual arguments must be separated by commas.

When you use the do statement to invoke another script, the scripts can exchange variable information. To pass a variable between scripts, declare the variable as public in the invoking script and as external in the invoked script.

do, like chain, compiles and runs a script source (.xws) file if there is no compiled version of the script, or if the date of the source file is more current than the date of the compiled version. Otherwise, do runs the compiled version of the script. Script names do not require an extension.

Versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information, see the arg function and the chain statement. Also refer to Chapter 3, "Declaring Variables, Arrays, Procedures, and Functions," for information on public and external variables; and to Chapter 4, "Interfacing with the Host, Users, and Other Scripts," for more information about invoking other scripts.

Examples

```
do "SCRIPT2"
```

In this example, a script called SCRIPT2 is invoked as a child script.

```
do "SCRIPT2", "CSERVE"
```

In this example, the argument CSERVE is passed to SCRIPT2.

Win

dosversion (function)

Use `dosversion` to get the DOS version number.

Format

```
x$ = dosversion
```

`dosversion` returns the DOS version number as a string.

Example

```
if dosversion < "3.0" then
   print "Incompatible version of DOS"
```

In this example, a message is displayed if the version of DOS is older than 3.0. ■

downloaddir (system variable)

Use downloaddir to read or set a directory other than the default directory for transfers and captures.

Format

```
downloaddir = <string>
```

downloaddir checks or sets a directory that is different from the dirfil directory for file transfers and data capture.

Normally transfers and captures are stored in the download directory specified by downloaddir. You can override the directory setting by setting a different path in dirfil. Note that some file transfer protocols do not provide the opportunity to specify the path; these protocols are autostart protocols, which immediately begin downloading the file. In this case, the file is placed in the current directory, which, in general, is not the same each time.

Versions of Crosstalk for Windows older than 2.0 do not support this variable.

For related information, see the dirfil system variable.

Example

```
downloaddir = "a:\DATA\FILDAT"
```

In this example, downloaddir is set to a:\DATA\FILDAT directory.

```
if exists(downloaddir+("\TEST.DAT")) then ...
```

In this example, the script tests whether the file TEST.DAT exists in the downloaddir directory.

Win

drive (statement)

Use `drive` to change the default disk drive.

Format

```
drive <string>
```

string must be an expression representing a valid disk drive. The default drive for all subsequent file operations will be set to the new drive.

Examples

```
drive "A:"
```

In this example, the drive is changed to "A:"

```
drive dname$
```

In this example, the drive is changed to the value contained in the variable `dname$`. ■

end (statement)

Use `end` to indicate the logical end of a script.

Format

```
end
```

`end` marks the logical end of a script. When an `end` statement is encountered, the following occurs:

- All variables associated with that script are discarded.
- All files opened by that script are closed.
- Execution of the script is terminated.
- If the script was invoked by a parent script, execution continues in the parent script.

Although it is a good programming practice to have an `end` statement at the physical end of the script source code as well as at the logical end of the source code, CASL accepts the physical end of the script as the logical end if no `end` statement is found.

For related information, see the `halt`, `quit`, and `terminate` statements.

Example

```
if not online then end
```

In this example, the script ends if it is not on line.

enhex (function)

Use enhex to convert a string of binary data to a string of ASCII characters in hexadecimal format.

Format

```
x$ = enhex(<string>)
```

enhex returns a string of ASCII characters that represent, in hexadecimal format, the data in *string*.

Since each byte in *string* is converted to a 2-byte hexadecimal representation, the string returned by enhex is twice as long as *string*.

Like entext and detext, enhex and dehex are complementary functions designed to permit the exchange of binary information over communications services that allow only 7-bit transfers (many of the electronic mail systems allow the transfer of only 7-bit ASCII information).

Binary data strings that have been converted with enhex require dehex to restore them to 8-bit binary format.

For related information, see the dehex, detext, and entext functions.

Examples

```
sendable = enhex(program_line)
```

In this example, enhex converts the binary string program_line to a string of ASCII characters and returns the result in sendable.

```
reply enhex(spread_sheet_line)
```

In this example, the script sends the result of the enhex conversion to the host.

enstore (function)

Use enstore to convert strings that may have embedded control characters into strings of printable ASCII characters.

Format

```
x$ = enstore(<string>)
```

In general, control characters are changed to caret-notation representation; that is, a Ctrl-G (bell) character is changed to ^G in the result. When you use the resulting string in a string operation such as a reply statement, the characters ^G are interpreted as Ctrl-G. The vertical bar (|) is used to represent Ctrl-M (CR).

enstore uses the backquote character (`) to protect any existing carets from later interpretation.

enstore is useful in script file management of passwords and other strings that often contain embedded control characters.

Strings that have been converted with the enstore function can be returned to their original form with the destore function.

For related information, see the destore function.

Examples

```
password = enstore("ALE" + chr(3))
```

In this example, the result of the enstore conversion is returned in password.

```
reply enstore(line_input_by_user)
```

In this example, the script sends the result of the enstore conversion to the host.

entext (function)

Use entext to convert a string of binary data to a string of printable ASCII characters.

Format

```
x$ = entext(<string>)
```

Like enhex and dehex, entext and detext are complementary functions designed to permit the exchange of binary information over communications services that allow only 7-bit transfers; many of the electronic mail systems allow the transfer of only 7-bit ASCII information.

Binary data strings that have been converted to ASCII with entext require the detext function to restore them to 8-bit binary format. The algorithm used by entext changes three 8-bit characters to four printable characters.

For related information, see the dehex, detext, and enhex functions.

Examples

```
sendable = entext(program_line)
```

In this example, the ASCII equivalent of the binary string program_line is assigned to sendable.

```
reply entext(spread_sheet_line)
```

In this example, spread_sheet_line is converted to ASCII characters and then sent to the host.

Win

environ (function)

Use environ to obtain the value of a DOS environment variable.

Format

```
x$ = environ(<string>)
```

environ returns the value of a specified operating system environment such as the path or the prompt.

string is not case-sensitive. A null string is returned if *string* is not found in the operating system environment.

Note: DOS environment variables must be set before you start Windows. Refer to your DOS manual for instructions on setting these variables. ■

Example

```
string dpath
dpath = environ("PATH")
```

In this example, the path setting is placed in the script's dpath variable. ■

eof (function)

Use eof to determine whether the end-of-file marker has been reached.

Format

```
x = eof[(<filenum>)]
```

eof returns true if the file specified in *filenum* is at the end of the file. eof returns false until the last record has been read; then it returns true.

If *filenum* is not specified, the file number defaults to the definput system variable.

In random files, eof returns true when the most recent get statement returns less than the requested number of bytes. get does not read past the end of the file.

In input (sequential) files, eof returns true when the most recent read or read line statement reads the last record in the file. The contents of the last record of a file depend on the method used to create it. Some applications place a Ctrl-Z (ASCII 26 decimal) character at the end of the file while other applications do not. Still other applications round out the file to a length evenly divisible by 128, either by writing multiple Ctrl-Z characters or by writing a single Ctrl-Z followed by whatever was in the rest of the output buffer on the previous write.

For related information, see the definput system variable and the get, read, and seek statements.

Example

```
string name
while not eof
   read name
   print name
wend
end
```

This code fragment reads strings from an already opened sequential file and prints them to the screen. When the end-of-file marker is reached, the `while`/`wend` loop is terminated, and the script ends.

eol (function)

Use eol to determine if a carriage-return/line-feed character, indicating the end of a line, was part of the data read during the last read statement.

Format

```
x = eol[(<filenum>)]
```

eol returns true if the last read statement encountered a carriage-return/line-feed (CR/LF) character.

filenum is the file number assigned to the file when it was opened. If *filenum* is not specified, the file number defaults to the definput system variable.

eol, like eof, indicates the status of a data file following a read operation; eol, however, works only on sequential input files, and reports whether the most recent read statement read the last field in the line (that is, encountered a CR/LF). Most applications use CR/LF to indicate the end of a line.

When reading comma-delimited ASCII files with read statements, use eol to ensure alignment of the file reading commands with the contents of the file, especially when the file in question was written using another application. The example provided shows this technique.

For related information, see the definput system variable and the read statement.

Example

```
string name
open input "names.dat" as 1
definput = 1
while not eof
    read name
    print name ;
    while not eol
        read name
        print " and " ; name ;
    wend
    print
wend
```

In this example, a file with a file number of 1 (one) is opened for input. The two `while/wend` loops control the read operations. The outer loop is set so that the file is read until the end-of-file marker is reached. Within each read operation, the inner loop ensures that all of the data through the end-of-line character is read and printed.

errclass (system variable)

Use errclass to check the type of the last error.

Format

```
x = errclass
```

errclass contains an integer reflecting the type of error that last occurred. It is zero if no error has occurred. errclass is not cleared when you check it. It remains unchanged until another error occurs.

For related information, see the errno system variable, the error function, and the trap compiler directive.

Example

```
trap on
send fname
trap off
if error then
   case errclass of
      45: goto file_tran_err
      26: goto call_fail_err
      default: goto other_err
   endcase
```

This example shows how to test for such things as file-transfer or call-failure errors after a script executes a file transfer command.

errno (system variable)

Use errno to check the specific type of the last error.

Format

```
x = errno
```

errno contains an integer reflecting the error number, within the errclass, for the error that last occurred. It is zero if no error occurs. errno is not cleared when checked. It remains unchanged until a different error has occurred.

For related information, see the errclass system variable, the error function, and the trap compiler directive.

Example

```
trap on
send fname
trap off
if error then E1 = errclass : E2 = errno
```

In this example, error trapping is turned on, a file transfer is attempted, and trapping is turned off. If an error occurred, E1 is set to the value in errclass and E2 is set to the value in errno.

error (function)

Use error to check for the occurrence of an error.

Format

```
x = error
```

error reports the occurrence of an error. It returns true if an error occurred and false if no error occurred. error is reset each time it is tested. If you want to continue to trap errors throughout the execution of the script, error must be cleared out (tested) after each error occurs.

When you use error with the trap compiler directive, you can direct program flow to an error handling routine.

error merely indicates that there has been an error. errclass and errno specify which error has occurred. errclass and errno are not cleared when tested.

Note: Fatal run-time errors cannot be trapped. ■

For related information, see the errclass and errno system variables and the trap compiler directive.

Example

```
trap on
compile "zark"
trap off
if error then print "compile failed"
```

In this example, error trapping is turned on and the script requests that zark be compiled. Then error trapping is turned off. If an error occurred, the script prints an error message.

exists (function)

Use `exists` to determine whether a file or subdirectory exists.

Format

```
x = exists(<string>)
```

`exists` returns true if the file specified in *string* exists, and false if it does not. Use `exists` only to check for files and subdirectories. It does not work for root directories.

string must be a legal file specification, and can contain drive specifiers, path names, and wild-card characters.

For related information, see the `fileattr` function.

Examples

```
print exists("XTALK.EXE")
```

In this example, either true or false is displayed, depending on the existence of the file `XTALK.EXE`.

```
if exists("C:\BIN") then
   print "BIN directory!"
```

In this example, a message is displayed if the directory BIN exists on the C drive.

```
if not exists(dat_file) then goto dat_error
```

In this example, the script branches to the label `dat_error` if the `dat_file` does not exist.

exit (statement)

Use `exit` to exit from a procedure.

Format

```
exit
```

When an `exit` statement is encountered, the procedure returns control to the statement following the one that called it.

For related information, see the `chain`, `do`, and `end` statements and the `proc ... endproc` procedure declaration.

Example

```
proc test takes integer x
   if x < 1 then exit
   print x; " seconds remaining."
endproc
```

In this example, the procedure `test` is called with the argument `x`. If `x` is less than 1, the procedure returns control to the statement following the one that called it. Otherwise, a message is displayed and then the procedure returns control when `endproc` is executed.

extract (function)

Use extract to return a string of characters that is removed from another string.

Format

```
x$ = extract(<string, wild [, where_int]>)
```

extract is, essentially, the opposite of the strip function; it returns the characters strip discards from a string.

wild can be either a string of the characters you want to return from *string* or it can be an integer bitmap of the Crosstalk character class(es) containing the characters you want returned. (See the class function earlier in this chapter for a list of classes.) Each character in *wild* is considered independently, and *wild* is case-sensitive.

where_int is an integer, with the following meanings :

0 Extract all occurrences in *string* of any character in *wild*.

1 Extract from the right side, stopping at the first occurrence of a character not in *wild*.

2 Extract from the left side, stopping at the first occurrence of a character not in *wild*.

3 Extract from both the right and left sides, stopping on each side at the first occurrence of a character not in *wild*.

extract is quite useful in analyzing lines read from word-processing text files, for counting leading zeros, and for editing user-entered strings.

Examples

```
print extract("0123456", "0", 2)
```

In this example, the script displays "0."

```
print extract("Sassafras", "as", 0)
```

In this example, the script displays "assaas."

false (constant)

Use `false` to set a boolean variable to logical false.

Format

```
x = false
```

`false` is always logical false. `false`, like its complement, `true`, exists as a way to set variables on and off. If `false` is converted to an integer, its value is 0 (zero).

For related information, see the `true`, `on`, and `off` constants.

Example

```
done = false
while not done
    ...
    ...
wend
```

In this example, the statements in the `while/wend` construct are repeated until `done` is `true`.

fileattr (function)

Use `fileattr` to return an attribute bitmap that describes the file's attributes.

Format

`x = fileattr[(<filename>)]`

If *`filename`* is used, `fileattr` returns the attributes of the file specified in *`filename`*.

If *`filename`* is not used, `fileattr` returns the attributes of the last file found by the `filefind` function.

The bitmap returned is the total of the possible attributes shown in Table 6-10.

Table 6-10. Bitmap values for the fileattr function

Hex	Dec	Attribute meaning
01h	1	A read-only Windows file or a locked Macintosh file.
02h	2	A hidden Windows or Macintosh file. The file is excluded from directory searches.
04h	4	A Windows system file. The file is excluded from directory searches. Note that this is not applicable for the Macintosh.
08h	8	The volume name of a Windows or Macintosh disk.
10h	16	A Windows directory or a Macintosh folder.
20h	32	A Windows or Macintosh archive bit. This bit indicates the file has been changed since it was last backed up.

filename must be a legal file specification. Path names are permitted; wild cards are not permitted. Some attribute bit combinations, though theoretically possible, may not be supported by your operating system.

For related information, see the `chmod` statement and the `filefind` function.

Example

```
print fileattr("xtalk.exe")
```

In this example, the script displays the attribute for the file `xtalk.exe`.

filedate (function)

Use `filedate` to return the date, in elapsed-day format, that the operating system assigned to a file.

Format

```
x = filedate[(<filename>)]
```

If *filename* is used, `filedate` returns the date of the file specified in *filename*.

If *filename* is not used, `filedate` returns the date of the last file found by the `filefind` function.

To simplify the comparison of file ages, the date is returned as an integer in elapsed-day format, giving the age of the file in days since the first day of January, 1900. To convert this to month-day-year format, use the `date` function.

filename must be a legal file specification. Path and drive specifiers are permitted; wild cards are not permitted.

For related information, see the `date` and `fileattr` functions.

Examples

```
print date(filedate("xtalk.exe"))
```

In this example, the script prints the date in day-month-year format.

```
file_age = filedate(file_string)
```

In this example, the date assigned to `file_string` is returned in `file_age`.

filefind (function)

Use `filefind` to check a file name.

Format

```
x$ = filefind[(<string> [, <integer>])]
```

`filefind` returns the full path name of a file matching the pattern specified in *`string`*. If *`string`* is not used, `filefind` returns the name of the next file in the directory that fits the last file specification given as *`string`*. If no such file is found, `filefind` returns the null string.

If both *`string`* and *`integer`* are used, `filefind` returns the name of the first file in the directory whose name matches *`string`* and whose attribute bitmap equals *`integer`*. (See the `fileattr` function earlier in this chapter for a list of possible attributes.) Note that the volume name attribute (08h or 8) is not supported.

`string` must be a legal file specification that can include drive specifiers and path names as well as wild-card characters.

For related information, see the `fileattr` function.

Example

```
x = filefind("*.*")
while not null(x)
   print x
   x = filefind
wend
```

In this example, the script displays a list of files in the current directory.

filesize (function)

Use `filesize` to check the size of a file.

Format

```
x = filesize[(filename>)]
```

If *`filename`* is used, `filesize` returns the size of the file specified in *`filename`*. If *`filename`* is not used, `filesize` returns the size of the file found by the most recent `filefind`.

`filename` must be a legal file specification that can contain drive specifiers and path names as well as wild-card characters.

For related information, see the `fileattr` and `filefind` functions.

Examples

```
progsize = filesize("XTALK.EXE")
```

In this example, the size of `XTALK.EXE` is returned in `progsize`.

```
print filesize
```

In this example, the script displays the size of the file found by the most recent `filefind`.

filetime (function)

Use `filetime` to determine the time a file was last updated, in seconds-elapsed format.

Format

`x = filetime[(<filename>)]`

If *`filename`* is used, `filetime` returns the time of the file specified in *`filename`*. If *`filename`* is not used, `filetime` returns the time of the file found by the most recent `filefind`.

To facilitate file-age comparisons, `filetime` is returned as an integer indicating the number of seconds past midnight since the file was created or last modified. To convert this to hours, minutes, and seconds, use the `time` function.

`filename` must be a legal file specification. Drive specifiers, path names, and wild-card characters are permitted.

For related information, see the `fileattr`, `filefind`, and `time` functions.

Examples

```
print time(filetime("xtalk.exe"))
```

In this example, the time that the file was last updated is displayed as hours, minutes, and seconds with AM or PM.

```
prog_age = filetime("xtalk.exe")
```

In this example, `filetime` returns the time the file was last updated in `prog_age`.

Win fncheck (function)

Use `fncheck` to check the validity of a file name specification.

Format

```
x = fncheck(<string>)
```

`fncheck` provides a quick way to parse file names. It returns a bitmap indicating the presence or absence of various file name parts such as the drive letter, path, name, file type extension, and wild cards.

The bitmap returned indicates which parts are present, as shown in Table 6-11.

Table 6-11. Bitmap values for the fncheck function

Hex	Dec	File name	Attribute/Meaning
01h	1	Drive	Found a colon.
02h	2	Path	Found a backslash.
04h	4	Extension	Found a dot.
08h	8	Wild card	Found a question mark.
10h	16h	Wild card	Found an asterisk.

The bitmap values are added together for every part of a file name that is found.

string should be a legal file name for the results to be meaningful.

For related information, see the `fnstrip` function.

Example

```
print fncheck(long_file_spec)
```

In this example, the various parts of the file name `long_file_spec` are displayed. ■

Win fnstrip (function)

Use `fnstrip` to return specified portions of a file name specification.

Format

`x$ = fnstrip(<string, specifier>)`

`fnstrip` provides a quick way to parse file names, breaking them down into component parts like the drive letter, path, and name.

`string` can be made up of the drive, path, name, and extension, as shown in the following example:

```
C:\xtalk\xtalk.exe
```

The parts of *`string`* that are returned are controlled by *`specifier`*, according to the bitmap values shown in Table 6-12.

Table 6-12. Bitmap values for the fnstrip function

Hex	Dec	Portion Returned
00h	0	Returns the full file name.
01h	1	Returns all except the drive designation.
02h	2	Returns the drive, file name, and extension.
03h	3	Returns the file name and extension.
04h	4	Returns the drive, path, and file name (no extension).
05h	5	Returns the path and file name (no extension).
06h	6	Returns the drive and file name (no extension).
07h	7	Returns the name only (no extension).

Add 8 to *`specifier`* to have the string returned in all uppercase characters; add 16 (decimal) to return the string in all lowercase characters.

string should be a legal file name for the results to be meaningful.

For related information, see the `fncheck` function.

Examples

```
print fnstrip(long_file_spec, 3)
```

In this example, the script displays the file name and extension.

```
progname = fnstrip(long_file_name, 7)
```

In this example, `fnstrip` returns only the file name.

```
U_Case_ProgName = ...
   fnstrip ("C:\XTALK4\xtalk.exe", 15)
```

In this example, `fnstrip` returns the file name in uppercase characters. ■

footer (system variable)

Use `footer` to define the footer used when printing from Crosstalk.

Format

`footer = <string>`

`string` can be any valid string expression. You can embed special characters in the string to print the current date, the time, and so on.

Crosstalk Mark 4 does not support this variable.

For related information, see the `header` system variables. Refer to your Crosstalk user's guide for additional information on footers.

Example

```
footer = "Date: " + date
```

In this example, the word `Date:` and the current date are assigned to `footer`.

for ... next (statements)

Use `for ... next` to perform a series of statements a given number of times while changing a variable.

Format

```
for <variable> = <startvalue> to <endvalue> ...
    [step <stepvalue>]
    ...
    ...
next [<variable>]
```

variable can be any integer or real variable. You do not have to declare the variable previously, but it is a good idea to do so.

startvalue, *endvalue*, and *stepvalue* are expressions; they can be any type of numeric expression. *startvalue* specifies the starting value for the counter and *endvalue* specifies the ending value.

The statements in the `for/next` construct are performed until the `next` statement is encountered. The value of *stepvalue* is then added to *variable*. (If you do not specify a step value, 1 is assumed.) Depending on whether *stepvalue* has a positive or negative value, one of the following occurs:

- If *stepvalue* is greater than or equal to 0 (zero), and, if *variable* is not greater than *endvalue*, the statements are repeated. However, if *startvalue* starts with a value greater than *endvalue*, the statements are not performed at all.
- If *stepvalue* is less than 0 (zero), and if *variable* is not less than *endvalue*, the statements are repeated. However, if *startvalue* starts with a value smaller than *endvalue*, the statements are not performed at all.

Caution: We recommend that you not change the value of *variable* within the construct. This can produce erroneous results. ■

You can nest `for/next` constructs; that is, you can place one construct inside another one. If you use nested constructs, be sure to use different variables in each construct. In addition, make sure that a nested construct resides entirely within another construct.

Versions of Crosstalk for Windows older than 2.0 do not support these statements.

Examples

```
for i = 1 to 10
   print i
next i
```

In this example, the `i` variable is incremented by 1 each time the `for/next` construct is repeated. With each repetition, the value of `i` is displayed on the screen.

```
for i = 10 to 1 step -1
   print i
next i
```

In this example, the `i` variable is decremented by 1 each time the `for/next` construct is repeated. With each repetition, the value of `i` is displayed on the screen.

```
for i = 0 to 100 step 5
   print i
next
```

In this example, the `i` variable is incremented by 5 each time the `for/next` construct is repeated. With each repetition, the value of `i` is displayed on the screen.

```
for i = 0 to 10
   print "Times table for "; i
   for j = 1 to 10
      print , i; " times "; j; " is: "; i * j
   next
   print
next
```

This is an example of nested `for/next` constructs. Multiplication tables for 1*1 through 10*10 are printed. Indentation is used here to show the relationship of the two constructs and for program readability.

freefile (function)

Use `freefile` to get the lowest available file number for the current session.

Format

```
x = freefile
```

`freefile` returns the number of the next available file number. It lets you write general-purpose scripts that do not require a specific file number. This is particularly valuable in a script that may form part of several other scripts.

The maximum number of file numbers available is 8. `freefile` returns zero if no file number is available.

Always store the results of the `freefile` function in a variable, since the value of the function will change every time a new file is opened.

For related information, see the `definput` system variable and the `close` and `open` statements.

Example

```
f = freefile
open input "z.dat" as #f
definput = f
```

In this example, the first line uses the `freefile` function to retrieve the next available file number and stores the number in the variable `f`. The next line opens a file called `z.dat` for input, and the last line saves the value of `f` in `definput`.

freemem (function)

Use `freemem` to find out how much memory is available.

Format

```
x = freemem
```

`freemem` returns the amount of memory that is available at the time the function is executed. The amount of available memory changes depending on the activity of other applications.

Examples

```
print freemem
```

In this example, the script displays the amount of unused memory.

```
if freemem > 64k then ...
```

In this example, the script tests whether available memory exceeds 64 KB.

freetrack (function)

Use `freetrack` to return the lowest unused track number for the current session.

Format

```
x = freetrack
```

`freetrack` returns the value of the next available track number. It lets you write general-purpose scripts that do not require a specific track number. This is particularly valuable in a script that may form part of several other scripts.

You can have any number of `track` statements active at one time, limited only by available memory. `freetrack` returns zero if no track numbers are available.

Always store the results of the `freetrack` function in a variable, since the value of the function will change every time a new track is used.

For related information, see the `track` function and the `track` statement.

Example

```
t1 = freetrack
track t1, space "system going down"
wait for key 27
if track(t1) then { bye : end }
```

In this example, the next available track number is assigned to `t1`. The `track` statement, using `t1`, watches for the specified string. Its occurrence is tested with the `track` function.

func ... endfunc (function declaration)

Use `func ... endfunc` to define and name a function.

Format

```
func <name> [([<type>] <argument> ...
   [, [<type>] <argument>]...)] returns <type>
   -----
   -----
endfunc
```

A function is similar to a procedure, but it returns a value. You must declare the type of the return value within the function definition and specify a return value before returning.

The arguments are optional. If arguments are included, you must use the same number and type of arguments in both the function and the statement that calls the function. The arguments are assumed to be strings unless otherwise specified.

Any variable declared within a function is local to the function. The function can reference variables that are outside the function, but not the other way around.

Functions can contain labels, and the labels can be the target of `gosub ... return` and `goto` statements, but such activity must be wholly contained within the function. If you reference a label inside a function from outside the function, an error occurs.

You can nest functions at the execution level; that is, one function can call another. However, you must not nest functions at the definition level; one function definition cannot contain another function definition.

You can use forward declarations to declare functions whose definition occurs later in the script. The syntax of a forward function declaration is the same as the first line of a function definition, with the addition of the `forward` keyword.

Forward declarations are useful if you want to place your functions near the end of your script. A function must be declared before you can call it; the forward declaration provides the means to declare a function and later define what the function is to perform.

The following format is used for a forward declaration:

```
func <name> [(<arglist>)] returns <type> ...
   forward
```

Functions can be in separate files. To include an external function in a script, use the `include` compiler directive.

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support function declarations.

For related information, see the `proc ... endproc` procedure declaration and the `include` compiler directive.

Examples

```
func calc(integer x, integer y) returns integer
     if x < y then return x else return y
endfunc
```

In this example, the integers `x` and `y` are the function arguments. The values of `x` and `y` are passed to the function when it is called. The function returns one or the other value depending on the outcome of the `if then else` comparison. If `x` is less than `y`, `x` is the return value; if `x` is not less than `y`, the value of `y` is returned.

```
func calc(integer x, integer y) returns ...
   integer forward

return_value = calc(3, 8)

func calc(integer x, integer y) returns integer
   if x < y then return x else return y
endfunc
```

In this example, the function `calc` is declared as a forward declaration. Then the function is called.

Note: For ease of programming, you do not have to supply the parameters in the actual function definition if you use a forward declaration. For instance, the foregoing example can also be written as follows:

```
func calc(integer x, integer y) returns ...
   integer forward

return_value = calc(3, 8)

func calc
   if x < y then return x else return y
endfunc ■
```

genlabels (compiler directive)

Use `genlabels` to include or exclude label information in a compiled script.

Format

```
genlabels {on | off}
```

`genlabels off` tells the script compiler to suppress label information in the compiled script. The resulting script is usually smaller if you use this directive. The default for the directive is `on`.

Note: You cannot use the `goto @<expression>` statement if your script contains the `genlabels off` compiler directive. ■

Versions of Crosstalk for Windows older than 2.0 do not support this compiler directive.

For related information, see the `genlines` compiler directive.

Example

```
genlabels off
```

In this example, `genlabels` is set to `off`.

genlines (compiler directive)

Use `genlines` to include or exclude line information in a compiled script.

Format

```
genlines {on | off}
```

`genlines off` tells the script compiler to exclude line information from the compiled script. The default for the directive is `on`.

Versions of Crosstalk for Windows older than 2.0 do not support this compiler directive.

For related information, see the `genlabels` compiler directive.

Example

```
genlines off
```

In this example, `genlines` is set to `off`.

get (statement)

Use get to read characters from a random file.

Format

```
get [# <filenum>, ] <integer>, <stringvar>
```

get reads *integer* bytes from the random file identified by *filenum*, and places the bytes read in the string variable *stringvar*. If *filenum* is not provided, the script processor uses the value in definput.

If the end-of-file marker is reached during the read, *stringvar* may contain fewer than *integer* bytes, and may even be null.

Each get advances the file I/O pointer by *integer* positions or to the end-of-file marker, whichever is first encountered.

To use the get statement, you must open the file in random mode and have already declared *stringvar*.

For related information, see the definput system variable, and the open, put, and seek statements.

Example

```
proc byte_check takes one_byte forward
string one_byte

get #fileno, 1, one_byte

while not eof(fileno)
   byte_check one_byte
   get #fileno, 1, one_byte
wend
```

This code fragment reads an already opened random file 1 byte at a time and calls a procedure to process the byte. This continues to happen until the end-of-file marker is reached.

go (statement)

Use go to establish communications with the host.

Format

```
go
```

go establishes a connection to the host and runs a logon script, if the session supports a logon script.

Note: To initiate this command using your Crosstalk application, choose Connection from the Action pull-down and then choose Connect. ■

For related information, see the bye, call, load, and quit statements.

Example

```
-- Let the user select the system
alert "Select Vax to call", "A", "B", "C", cancel
-- Load the specified profile
case choice of
   1       : load "vaxa"
   2       : load "vaxb"
   3       : load "vaxc"
   default : end
endcase
-- Go online
go
```

This example shows how to use the case/endcase construct to handle user input in the alert dialog box. If the case statement default option is executed, the script ends. Otherwise, the script loads the appropriate session and uses the go statement to establish a connection to the host.

gosub ... return (statements)

Use `gosub` to transfer program control temporarily to a subroutine. Use `return` to return control to the calling routine.

Format

```
gosub <label>

<label>:
...
...
return
```

label must be the name of a subroutine label. The subroutine must end with a `return` statement.

Subroutines are helpful when you need to execute the same statements many times in a script. You can use subroutines as many times as needed, and you can use the `gosub` statement in a subroutine to pass control to other subroutines. You can have up to 8 nested subroutines.

When a `gosub` statement is encountered, the script branches to *label*. When a `return` statement is encountered, program control returns to the statement after the one that called the subroutine. A subroutine can have more than one `return` statement.

Subroutines can appear anywhere in a script, but it is a good programming practice to put all of your subroutines together, usually at the end of the script.

For related information, see the `goto`, `label`, and `pop` statements.

Example

```
text = "Hello, there."
gosub print_centered
end
label print_centered
   l = length(text)
   if l = 0 then return
   print at ypos, (80/2)-(length(text)/2), text
   return
```

This example shows a subroutine called `print_centered` that displays a string called `text`, centered on the screen in the default window.

goto (statement)

Use `goto` to branch to a label or expression.

Format

```
goto <label>
goto @<expression>
```

label must be the name of a program label.

expression can be any string expression that represents a label in the script. If you specify an expression, you must precede the expression with the 'at' sign (@), which forces the expression to be evaluated at run time.

When a `goto` statement is encountered in a script, the script branches to *label*.

Note: If you use the `goto @<expression>` form of this statement in your script, you cannot use the `genlabels off` compiler directive. ■

For related information, see the `gosub ... return` and `label` statements.

Examples

```
goto main_menu
```

In this example, the script branches to the label `main_menu`.

```
goto @"handle_" + xvi_keyword
```

In this example, the script branches to the specified expression.

grab (statement)

Use `grab` to send the contents of a session window to the snapshot file.

Format

```
grab
```

`grab` takes a snapshot of the current window, putting an image of the screen in the snapshot file.

Example

```
grab
```

halt (statement)

Use `halt` to stop script execution.

Format

```
halt
```

When a `halt` statement is encountered in a script, the script is immediately stopped. If there is a related parent script, it terminates also.

Note: To stop a script using your Crosstalk application, choose Stop from the Script pull-down. ■

For related information, see the `end` statement.

Example

```
if not online then halt
```

In this example, the script stops executing if it is not on line to the host.

header (system variable)

Use header to define the header used when printing from Crosstalk.

Format

```
header = <string>
```

string can be a any valid string expression. You can embed special characters in the string to print the current date, the time, and so on.

This variable is not supported by Crosstalk Mark 4.

For related information, see the footer system variable. Refer to your Crosstalk user's guide for more information on headers.

Example

```
header = "Printed using the " + description ...
   + " entry."
```

In this example, the specified string is assigned to header.

hex (function)

Use hex to convert an integer to a hexadecimal string.

Format

```
x$ = hex(<integer>)
```

hex returns a string giving the hexadecimal representation of *integer*. If *integer* is between 0 (zero) and 65,535, the string is 4 characters long; otherwise, it is 8 characters long.

Example

```
print hex(32767)
```

In this example, the script displays the hexadecimal equivalent of the integer 32,767.

hide (statement)

Use `hide` to reduce a session window to an icon.

Format

```
hide
```

This statement reduces a Crosstalk session window to an icon.

For related information, see the `show`, `minimize`, and `maximize` statements.

Example

```
hide
```

hideallquickpads (statement)

Use `hideallquickpads` to hide all of the QuickPads for the current session.

Format

```
hideallquickpads
```

This statement hides all of the QuickPads for the current session.

Note: The QuickPads for the session must already be loaded using the `loadquickpad` or `loadallquickpads` statement. ■

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information, see the `hidequickpad`, `loadquickpad`, `loadallquickpads`, `showallquickpads`, and `showquickpad` statements.

Example

```
hideallquickpads
```

hidequickpad (statement)

Use `hidequickpad` to hide the specified session QuickPad.

Format

```
hidequickpad <string>
```

This statement hides the session QuickPad specified in *string*.

Note: The QuickPad for the session must already be loaded using the `loadquickpad` or `loadallquickpads` statement. ■

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information, see the `hideallquickpads`, `loadquickpad`, `loadallquickpads`, `showallquickpads`, and `showquickpad` statements.

Example

```
hidequickpad "sessA"
```

In this example, the QuickPad identified as `sessA` is hidden.

hms (function)

Use hms to return a string in a time format.

Format

```
x$ = hms(<integer [, time_type]>)
```

hms converts *integer* to a string in any one of a number of time formats. *integer* is a number expressed in tenths of seconds, the same unit of time CASL uses for systime and tick.

time_type is a value that controls the format returned. It defaults to zero.

Table 6-13 shows examples for hms(300011, *time_type*) and hms(101, *time_type*).

Table 6-13. Bitmap values for the hms function

Hex	Decimal	30011 format	101 format
00h	0	8:20:01.1	0:00:10.1
01h	1	8:20:01.1	10.1
02h	2	8:20:01	0:00:10
03h	3	8:20:01	10
04h	4	8h20m1.1s	0h0m10.1s
05h	5	8h20m1.1s	10.1s
06h	6	8h20m1s	0h0m10s
07h	7	8h20m1s	10s

For related information, see the systime function.

Examples

```
print hms(300011)
```

In this example, the script displays the time.

```
print hms(systime, 6)
```

In this example, `hms` uses a decimal 6 value to control the format of the value that is displayed.

if ... then ... else (statements)

Use `if ... then ... else` to control program flow based on the value of an expression.

Format

```
if <expression> then

<statement group> ...

[else <statement group>]
```

expression is required, and can be any type of numeric, string, or boolean expression; or it can be a combination of numeric, string, and boolean expressions connected with logical operators such as `or`, `and`, or `not`. *expression* must logically evaluate to either `true` or `false`. Integers need not be explicitly compared to zero, but strings must be compared to produce a `true`/`false` value. For example, the following values evaluate logically to a `true` condition:

```
1
1 = 1
1 = (2-1)
"X" = "X"
"X" = upcase("x")
```

The following conditions evaluate to a `false` condition:

```
0
1 - 1
1 = 2
"X" = "Y"
```

`then` specifies the statement to perform if the expression or combination of expressions is `true`. `then` must appear on the same line as the `if` with which it is associated, as shown in the following example:

```
if done = true then
   print "Done!"
```

`else` specifies an optional statement to perform if the expression is not true. Each `else` matches the most recent unresolved `if`.

Blank lines are not allowed within a `then/else` statement group. If you want to place blank lines in the `then/else` statement group (for example, for the purpose of making the text more readable) use braces (`{}`) to enclose a series of statements.

Examples

```
label ask
   integer user_choice
   input user_choice
   if user_choice = 1 then
      print "Choice was 1." else
      if user_choice = 2 then
         print "Choice was 2." else goto ask
```

This example shows how to nest `if` statements in other `if` statements.

```
if choice = 1 then print "That was 1." : alarm
```

This example shows how to specify multiple statements after an `if` statement. In this case, the `print` and `alarm` statements are performed only if `choice` equals `1`.

```
if choice=1 or choice=2 then print "One or two."
if online and (choice=1) then print "We're OK."
if x=1 or (x=2 and y<>9) then ...
```

These three examples show how to specify multiple conditions in an `if then` statement. If the order in which the conditions are evaluated is important, use parentheses to force the order, as shown in the second and third examples.

`if then` statements can become quite complex. To make them easier to read, you can continue them over several lines by using braces to indicate a series of statements. The following example shows how to use braces:

```
if track(1) then
{
    bye
    wait 8 minutes
    call "megamail"
    end
}
```

You can also use braces to denote the `then` with which an `else` should be associated, as shown in the following example:

```
if x then { if y then a } else b
```

include (compiler directive)

Use include to include an external file in your script.

Format

```
include <filename>
```

include is a compile-time directive, normally used to include a source file of commonly used procedures and subroutines in a script. *filename* is required and must be the name of an existing file containing CASL language elements. For the Windows environment, if a file extension is omitted, .XWS is assumed.

include does not include the same file more than once during compilation.

For related information, see the chain and do statements, the func ... endfunc function declaration, and the proc ... endproc procedure declaration.

Example

```
include "myprocs"
```

In this example, the external file myprocs is included in the script.

inject (function)

Use `inject` to return a string with some characters changed.

Format

```
x$ = inject(<old_string, repl_string [, integer]>)
```

`inject` creates a new character string based on *`old_string`* but replacing part of *`old_string`* with the characters in *`repl_string`*, beginning at the first character in *`integer`*. The resulting string is the same length as *`old_string`*. *`old_string`* is unchanged.

`repl_string` is truncated if it is too long. If *`integer`* is omitted, the first character position is assumed.

`old_string` cannot be null, and *`integer`* must be in the range of `1 <= integer <= length(old_string)`.

Examples

```
print inject("XWALK.EXE", "T", 2)
```

In this example, the `W` in `XWALK.EXE` is changed to a `T` and the result is displayed.

```
dog_name = inject("xido", "F")
```

In this example, the `x` in `xido` is changed to an `F` and the result is stored in `dog_name`.

inkey (function)

Use `inkey` to return the value of a keystroke.

Format

```
x = inkey
```

`inkey` tests for keystrokes "on the fly," that is, without stopping the script to wait for a keystroke. This is particularly useful if you want to check for a keystroke while performing other operations.

`inkey` returns the ASCII value (0–255 decimal) of the key pressed for the printable characters and a special Crosstalk stroke value for the arrow keys, function keys, and special purpose keys. The keyboard keys and their corresponding numbers are listed in Table 6-14.

Table 6-14. Keyboard keys and their corresponding numbers

Keyboard key	Key number
F1 to F10	1025 to 1034
SHIFT-F1 to SHIFT-F10	1035 to 1044
CTRL-F1 to CTRL-F10	1045 to 1054
ALT-F1 to ALT-F10	1055 to 1064
↑	1281
↓	1282
←	1283
→	1284
HOME	1285
END	1286

continued

Table 6-14. Keyboard keys and their corresponding numbers (cont.)

Keyboard key	Key number
PGUP	1287
PGDN	1288
INS	1297
DEL	1298

If no keystroke is waiting, `inkey` returns zero. To clear the keyboard buffer before testing for a keystroke, use the following code:

```
while inkey : wend
```

`inkey` clears the keystroke from the keyboard buffer. If the key is important, store it in a variable, and then test the variable as shown in the following example:

```
x = inkey
if x <> 0 then ...
```

To make the user press the ESC key so the script can continue, use the following example:

```
print at 0, 0 , "Press ESC";
while inkey <> 27
wend
```

Examples

```
if inkey then end
```

In this example, the script ends if any key is pressed.

```
while not eof(file1) and inkey <> 27 ...
```

In this example, a task is performed while the end-of-file marker has not been reached and the ESC key is not pressed.

input (statement)

Use `input` to accept input from the keyboard.

Format

```
input <variable>
```

`variable` is required, and can be any type of numeric or string variable. You can use the backspace key to edit input.

Example

```
input username
```

In this example, the data in `username` is accepted by the script.

inscript (function)

Use `inscript` to check the labels in a script.

Format

```
x = inscript(<expression>)
```

`inscript` uses *`expression`* to check for the presence of a particular label in a script. The value returned is `true` if *`expression`* is a label in the currently running script, `false` otherwise. *`expression`* must be a string.

Note: The `genlabels` compiler directive must be `on` for this function to be effective. ■

For related information, see the `label` statement.

Example

```
if inscript("HA_" + user_input) then ...
```

In this example, the script tests for the presence of the specified label.

insert (function)

Use `insert` to return a string with some characters added.

Format

```
x$ = insert(<old_string, insert_string [, integer]>)
```

`insert` creates a new character string based on *`old_string`* by adding the characters in *`insert_string`* at the *`integer`* character position. The length of the resulting string is the combined length of *`old_string`* and *`insert_string`*. *`old_string`* is unchanged.

If *`integer`* is omitted, the first character position is assumed.

`old_string` cannot be null, and *`integer`* must be in the range of `1 <=` *`integer`* `<= length(`*`old_string`*`)`.

Examples

```
print insert("XALK.EXE", "T", 2)
```

In this example, the script inserts a `T` in the second position of `"XALK.EXE"` and displays the result.

```
dog_name = insert("ido", "F")
```

In this example, an `F` is inserted in the first position of `"ido"` and the result is stored in `dog_name`.

instr (function)

Use instr to return the position of a substring within a string.

Format

```
x = instr(<string, sub_string [, integer]>)
```

instr reports the position of *sub_string* in *string* starting its search at character *integer*. If *integer* is not given, the search begins at the first character. If *sub_string* is not found within *string*, zero is returned.

```
instr("Sassafras", "a")      returns 2
instr("Sassafras", "a", 3)   returns 5
```

instr can be used within a loop to detect the presence of a character you want to change to another character. The following code fragment expands the tab characters, which some text editors automatically embed in lines of text.

```
tb=chr(9)
t=instr(S, tb)
while t
   s=left(S, t-1) + pad("", 9-(t mod 8)) + mid(S, t+1)
   t=instr(S, tb)
wend
```

Examples

```
dog_place = instr("Here, Fido!", "Fido")
```

In this example, the substring Fido is found in position 7 of the string and the result is returned in dog_place.

```
if instr(fname, ".") = 0 then
   fname = fname + ".XWS"
```

In this example, the script looks for the presence of the file extension for fname. If an extension delimiter (.) is not found, the extension is added.

intval (function)

Use `intval` to return the numeric value of a string.

Format

```
x = intval(<string>)
```

`intval` returns an integer; it evaluates *`string`* for its numerical meaning and returns that meaning as the result. Leading white-space characters are ignored, and *`string`* is evaluated until a non-numeric character is encountered.

The script language is quite flexible as to the number base (decimal or hexadecimal) in question; terminate *`string`* with an "h" if it is hex, or "k" if it is decimal (k is for kilo bytes, so 1k = 1024).

A hexadecimal string cannot begin with an alphabetic character. If the string does not start with a numeric character, place a 0 (zero) at the beginning of the string.

The characters that have meaning to the `intval` function are: "0" through "9", "a" through "f", "A" through "F", "h", "H", "b", "B", "o", "O", "q", "Q", "k", "K", and "-".

For related information, see the `val` function.

Example

```
num = intval(user_input_string)
```

In this example, `user_input_string` is converted to an integer and returned in `num`.

jump (statement)

The `jump` statement, which is a synonym for the `goto` statement, is supported only for backward compatibility. Refer to `goto` earlier in this chapter.

kermit (statement)

Use kermit to send a command to the Kermit Command Processor (KCP).

Format

```
kermit <command>
```

command can be any one of the following:

```
"get <filename>"
"send <filename>"
"finish"
```

The kermit statement sends one of three possible commands to the KCP. *filename* is the name of the file(s) to be sent or received; this parameter is required only for the get and send commands.

get, send, and finish are the valid kermit commands. Table 6-15 explains these commands.

Table 6-15. Commands for the kermit statement

Option	Explanation
get	Requests the specified file(s) from the host server. This command is valid only when the host Kermit server is active. *filename* must be the name of an existing file on the host system.
send	Sends the file(s) specified by *filename* to the host. You can use wild-card characters to specify multiple files.
finish	Terminates the Crosstalk KCP and returns the host Kermit server to its command state. For some hosts, it is necessary to send a carriage return to enable the host to redisplay its Kermit prompt.

Note: To access the KCP using your Crosstalk application, choose File Transfer from the Action pull-down and then choose Command Processor. ■

Versions of Crosstalk for Windows older than 2.0 support additional KCP commands. Crosstalk Mark 4 does not support this statement.

Examples

```
kermit "get memo.txt"
```

In this example, the `kermit` statement uses the `get` command to request the file `memo.txt` from the host.

```
kermit "send *.txt"
```

In this example, the `kermit` statement uses the `send` command to send all files with a .txt extension to the host.

```
kermit "finish"
```

In this example, the `kermit` statement uses the `finish` command to terminate the Crosstalk KCP and return the host KCP server to command state.

keys (system variable)

Use keys to read or set the Keymap file for the current session.

Format

```
keys = <string>
```

keys specifies the name of the Keymap file for the current session. This file is created using the Keyboard Editor.

Versions of Crosstalk for Windows older than 2.0 do not support this variable.

Example

```
if keys = "MYKEYS" then
```

In this example, the script tests whether the content of keys is "MYKEYS."

label (statement)

Use `label` to specify a named reference point in a script file.

Format

`label <labelname>`

labelname can be made up of almost any printable characters.

Labels are used in scripts to provide a means of identifying a particular line in a program.

Do not use reserved words or special characters as a label name.

For related information, see the `goto` and `gosub ... return` statements.

Example

```
label ask
input user_choice
if user_choice = 1 then
   print "Choice = 1."
return
```

In this example, the `label` statement defines the location of the `ask` subroutine.

left (function)

Use `left` to return the left portion of a string.

Format

`x$ = left(<string [, integer]>)`

`left` returns the leftmost *integer* characters in *string*. If *integer* is not specified, the first character in *string* is returned. If *integer* is greater than the length of *string*, then *string* is returned.

For related information, see the `mid`, `right`, and `slice` functions.

Examples

```
dog_name = left("Fidox", 4)
```

In this example, `left` returns "`Fido`."

```
print left(long_string, 78)
```

In this example, the first 78 characters of `long_string` are displayed.

```
reply left(dat_rec, 24)
```

In this example, the first 24 characters of `dat_rec` are sent to the host.

length (function)

Use `length` to return the length of a string.

Format

```
x = length(<string>)
```

CASL allows strings of up to 32,767 characters; therefore, `length` always returns integers in the range of `0 <= length(<string>) <= 32767`. `length` returns zero if *`string`* is null.

Examples

```
print length(dog_name), dog_name
```

In this example, the script displays both the length of the string `dog_name` and the contents of the string.

```
if length(txt_ln) then reply txt_ln
   else reply "-"
```

In this example, the script sends the contents of `txt_ln` to the host if `txt_ln` contains data. Otherwise, the script sends a dash to the host.

linedelim (system variable)

Use `linedelim` to define a string to be sent after each line of text in an `upload` statement.

Format

`linedelim = <string>`

Most information services interpret a carriage return (CR) (ASCII decimal 13) as meaning "end of line," and that character is the default for `linedelim`. Some applications, however, require a special character at the end of each line. When this is the case, you can assign a special character to `linedelim`; Crosstalk will send that character instead of a CR at the end of each line when uploading text.

The most likely character to use for `linedelim`, other than a CR, is either a Ctrl-C (ASCII decimal 3) or a line-feed (ASCII decimal 10). `linedelim` cannot exceed 8 characters.

If you need to send a control character, use a caret (^), followed by the character. For example, Ctrl-C would be entered as ^C.

For related information, see the `upload` statement.

Example

```
linedelim = chr(3)
```

In this example, `linedelim` is set to a Ctrl-C.

linetime (system variable)

Use `linetime` to control the maximum time to wait before uploading the next line of text.

Format

```
linetime = <integer>
```

`linetime` is a fall-back parameter for the `lwait` statement and overrides the `lwait` parameter if the `lwait count`, `lwait echo`, or `lwait prompt` condition is not satisfied in *integer* seconds. The maximum value of *integer* is 127.

If *integer* is zero, or if `lwait` is `none`, `linetime` is disabled.

When *integer* seconds have elapsed since the last text line was sent, the next line is sent regardless of the satisfaction of the `lwait` statement.

This is most useful when sending long files over a questionable phone line. For example, suppose `lwait` is set to `prompt ":"`, and a long text file is being uploaded to a host system. If, for some reason, one of the ":" characters gets lost coming back, Crosstalk will wait forever for that colon character, unless `linetime` is set to some reasonable value, like 10 seconds.

For related information, see the `lwait` statement.

Example

```
linetime = 10
```

In this example, the maximum time to wait before uploading the next text line is 10 seconds.

load (statement)

Use `load` to load new settings into a session.

Format

```
load <string>
```

`load` is similar to the `call` statement, except that `call` attempts to establish a connection while `load` does not.

For related information, see the `call` statement.

Examples

```
load "cserve"
```

In this example, the script loads new settings from a session file called CSERVE.

```
string entry_name
entry_name = "source"
load entry_name
```

This example shows how to define a variable, set the variable to a session name, and then load the session settings using the variable.

loadquickpad (statement)

Use `loadquickpad` to open and display a QuickPad for the current session.

Format

```
loadquickpad <string>
```

This statement opens and displays the QuickPad specified in *string*. If the QuickPad is already open, the statement displays the QuickPad.

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information, see the `unloadquickpad` statement.

Example

```
if online then
   loadquickpad "apad"
```

In this example, the QuickPad named `apad` is activated.

loc (function)

Use `loc` to return the position of the file pointer.

Format

```
x = loc[(<filenum>)]
```

`loc` returns the byte position of the next read or write in a random file.

If *filenum* is not given, the default file number is assumed. You can set the default file number by using the `definput` system variable.

This function is valid only for files opened in `random` mode.

For related information, see the `definput` system variable and the `open` and `seek` statements.

Examples

```
print loc(1)
```

In this example, the script displays the location of the input/output pointer for file number 1.

```
if loc(1) = 8k then
   print "Eight kilobytes read."
```

In this example, the script prints the specified phrase if the file pointer is 8 KB into the file.

lowcase (function)

Use `lowcase` to convert a string to lowercase letters.

Format

```
x$ = lowcase(<string>)
```

`lowcase` converts only the letters A–Z to lowercase characters; numerals, punctuation marks, and notational symbols are unaffected.

`lowcase` is useful for testing string equivalence since it makes the string case-insensitive.

For related information, see the `upcase` function.

Examples

```
print "Can't find "; lowcase(fl_name)
```

In this example, the script displays a phrase that contains a file name in lowercase letters.

```
if lowcase(password) = "secret" then ...
```

In this example, the script takes some action if the contents of `password` match `"secret."`

lprint (statement)

Use `lprint` to send text to the system printer.

Format

```
lprint [<item>] [{,|;} [<item>]] ... [;]
```

lprint can take any item or list of items, including integers, strings, and quoted text, separated by semicolons or commas. *item* can be either an expression to be printed, the `EOP` keyword, or the `EOJ` keyword. If the items in the list are separated by semicolons, they are printed with no space between them; if separated by commas, they are printed at the next tab position. If no *item* is provided, a blank line is printed.

A trailing semicolon at the end of the `lprint` statement causes the statement to be printed without a carriage return. This is useful when you want to print something immediately after the statement on the same line.

Text is buffered in a print spooler. `EOP` indicates that printing should continue on another page. `EOJ` indicates the end of the print job; that is, the print spooler can now send the data to the printer. If your script ends without executing an `lprint EOJ`, the script processor executes one for you.

Examples

```
lprint "This is being sent to the printer."
```

This example shows how to print a simple phrase.

```
lprint "There's no carriage return after this.";
```

This example shows how to suppress a carriage return.

```
lprint "Current protocol is " ; protocol
```

This example shows how to print two phrases with no space between them.

```
lprint "Hello, " , name$
```

This example shows how to print a phrase followed by an automatic tab to `name$`.

lwait (statement)

Use `lwait` to control ASCII text uploads by pacing lines.

Format

```
lwait  {none | echo | prompt <charstring> | ...
   count <integer> | delay <real or integer>}
```

`lwait` controls text uploads by defining the condition to be met before the next line of text can be sent. The `lwait` parameters are explained in Table 6-16.

Table 6-16. Parameters for the lwait statement

Parameter	Explanation
`none`	Use this option with systems that are designed to accept full-speed uploads, such as electronic mail systems, or if you have used the `cwait` statement. Refer to `cwait` earlier in this chapter for a description of the statement.
`echo`	Use this to wait until the host sends a carriage return (CR) before sending the next line.
`prompt`	Use this to wait until the prompt string *`charstring`* is received from the host, and then send the next line. For example, some systems send a colon (:) when they are ready for the next line of text. In this case, you should use `lwait prompt ":"` to tell Crosstalk to wait for the colon. The maximum prompt length is 8 characters.
`count`	Use this to wait to receive *`integer`* characters from the host, and then send the next line. This is useful when sending text to systems that send a variable prompt (such as a line number) before accepting the line of text. The maximum value for *`integer`* is 255.

continued

Table 6-16. Parameters for the lwait statement (cont.)

Parameter	Explanation
`delay`	Use this to wait *`integer`* (or *`real`*) seconds before sending the next line. This is most useful in cases where the host system won't accept text at full speed and doesn't send any type of prompt. The maximum value for *`integer`* is 25.

You can use one of these `lwait` parameters with the `linetime` system variable to control the speed of text uploads to host computers. Note that only one parameter can be in effect at any one time.

The `lwait` statement is effective only when you are on line, but you can set the parameters when you are on line or off line.

For related information, see the `cwait` statement and the `linetime` system variable.

Examples

```
lwait echo
```

In this example, the script waits until the host sends a carriage return and then it sends the next line of text.

```
lwait prompt ":"
```

In this example, the script waits until a colon (:) is received from the host and then it sends the next line.

```
lwait count 3
```

In this example, the script waits until the host sends 3 characters before sending the next line.

match (system variable)

Use match to check the string found during the last wait or watch statement.

Format

```
x$ = match
```

match returns the most recent string for which the script was watching or waiting (up to 512 characters). For example, if the last wait or watch was looking for a keystroke, match returns the string value of the key pressed.

Use match only when you are on line.

For related information, see the wait and watch ... endwatch statements.

Example

```
wait 1 minute for "Login", "ID", "Password"
case match of
   "Login": reply logon
   "ID": reply userid
   "Password": reply password
endcase
```

In this example, the script waits up to 1 minute for the host to send a prompt. The script then uses the case/endcase construct to determine what response to send to the host.

max (function)

Use max to return the greater of two numbers.

Format

```
x = max(<number1>, <number2>)
```

max compares two numbers and returns the greater of the two.

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 support the max operator; they do not support the max function. This version of CASL supports only the max function.

For related information, see the min function.

Example

```
integer a, b, c
a = 1
b = 2

c = max(a, b)
```

In this example, the script declares three variables as integers and initializes two of them. Then it uses the max function to compare the integers a and b and returns the greater of the two in c. The result is c = 2.

maximize (statement)

Use `maximize` to enlarge the Crosstalk application window to full screen size.

Format

```
maximize
```

This statement lets you maximize the Crosstalk application window to its largest size. The `maximize` statement performs the same function as the Maximize option from the application window's Control menu.

Crosstalk Mark 4 does not support this statement.

For related information, see the `minimize`, `move`, `restore`, and `size` statements.

Example

```
maximize
```

message (statement)

Use `message` to display a user-defined message on the status bar of the screen.

Format

`message [<string>]`

`message` without an argument returns the information line to system control.

Crosstalk Mark 4 does not support this statement.

Examples

```
message "Logging in -- Please wait"
```

This `message` statement displays a simple message.

```
message "Today " + curday
```

This `message` statement displays a phrase as well as the current day.

mid (function)

Use mid to return the middle portion of a string.

Format

```
x$ = mid(<string>, <start> [, <len>])
```

mid returns the middle portion of *string* beginning at *start*, and returns *len* bytes. If *len* is not specified, or if *start* plus *len* is greater than the length of *string*, then the rest of the string is returned.

Examples

```
dog_name = mid("Here, Fido, here boy!", 7, 4)
```

In this example, mid returns "Fido" in dog_name.

```
if mid(fname, 2, 1) = ":" then dv = left(fname, 1)
```

In this example, dv is assigned the first character in fname if the second character in fname is a colon.

min (function)

Use `min` to return the lesser of two numbers.

Format

`x = min(<number1>, <number2>)`

`min` compares two numbers and returns the lesser of the two.

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 support the `min` operator; they do not support the `min` function. This version of CASL supports only the `min` function.

For related information, see the `max` function.

Example

```
integer a, b, c
a = 1
b = 2

c = min(a, b)
```

In this example, the script declares three variables as integers and initializes two of them. Then it uses the `min` function to compare the integers `a` and `b` and returns the lesser of the two in `c`. The result is `c = 1`.

minimize (statement)

Use `minimize` to reduce the Crosstalk application window to an icon.

Format

```
minimize
```

This statement lets you minimize the Crosstalk application window. The `minimize` statement performs the same function as the Minimize option from the application window's Control menu.

Crosstalk Mark 4 does not support this statement.

For related information, see the `maximize`, `move`, `restore`, and `size` statements.

Example

```
minimize
```

mkdir (statement)

Use `mkdir` to create a new subdirectory.

Format

`mkdir <directory>`

directory must be a string expression containing a valid directory name.

An error occurs if *directory* or a file with the same name as the one you have specified for the directory already exists.

Note: You can also use the abbreviation `md` for this statement. ■

Examples

```
mkdir "C:\XTALK\FILE"
```

In this example, the script creates a directory called `FILE` under the C:\XTALK directory.

```
mkdir "FILE"
```

In this example, the script creates a subdirectory called `FILE` under the current drive and directory.

mkint (function)

Use mkint to convert strings to integers.

Format

```
x = mkint(<string>)
```

The mkint and mkstr functions are mirror-image conversion functions that allow you to store 32-bit integers in 4-byte strings.

Use mkint to convert strings to integers when you read the file.

Example

```
get #1, 4, a_string : a_num = mkint(a_string)
```

In this example, the get statement reads 4 bytes of data from the file with file number #1 and stores the bytes in a_string. Then the mkint function converts the data in a_string to an integer and stores the result in a_num.

mkstr (function)

Use mkstr to convert integers to strings for more compact file storage.

Format

```
x$ = mkstr(<integer>)
```

The mkint and mkstr functions are mirror-image conversion functions that allow you to store 32-bit integers in 4-byte strings.

Use mkstr to convert integers to strings for compact storage in a file.

Examples

```
print mkstr(65), mkstr(6565), mkint("A")
```

In this example, mkstr converts 65 and 6565 to strings and mkint converts "A" to its equivalent integer value.

```
put #1, mkstr(very_big_num)
```

In this example, the mkstr function converts very_big_num to a string, and the put statement writes the string to a file.

Win

move (statement)

Use move to move the Crosstalk window to a new location on the screen.

Format

```
move <x>, <y>
```

This statement moves the Crosstalk window to the location specified by *x* and *y*, in pixels. The range of coordinates depends on the video hardware used.

For related information, see the maximize, minimize, restore, and size statements.

Examples

```
move 2, 30
```

This example shows how to move the window to column 2, row 30.

```
move x, y
```

In this example, the script moves the window to the location defined by the x and y variables. ■

name (function)

Use `name` to get the name of the current session.

Format

```
x$ = name
```

`name` returns the name of the current session.

The name of the session appears on the session window Title Bar.

Example

```
if name = "cserve" then go
```

In this example, if the name of the session is CSERVE, dial the modem.

netid (system variable)

Use `netid` to read or set a network identifier for the current session.

Format

`netid = <string>`

`netid` sets and reads the network address associated with the current session. The `netid` is limited to 40 characters and is optional.

Note: To set the equivalent parameter using your Crosstalk application, choose Session from the Settings pull-down. Then choose the General icon and modify the Network ID parameter. ■

Example

```
netid = "CIS02"
```

In this example, `netid` is set to the specified string.

new (statement)

Use new to open a new session.

Format

```
new [<filename>]
```

This statement opens the session specified in *filename*. If *filename* is omitted, an untitled session is opened.

For related information, see the call and load statements.

Example

```
new "CSERVE"
```

nextchar (function)

Use nextchar to return the character waiting at the communications device.

Format

```
x$ = nextchar
```

nextchar returns the character waiting at the communications device. If no character is waiting, nextchar returns a null string and processing continues.

The nextchar function clears the current character from the device; if you want to retain the character, store it in a variable and then test the variable.

Note that nextchar returns a string, while inkey returns an integer.

The following code uses the nextchar and inkey functions to get characters from the device and the keyboard, respectively:

```
/* The terminal, assumes full duplex host. */
string nchar
integer kpress
while kpress <> 27
   nchar = nextchar
   if not null(nchar) then print nchar;
   kpress = inkey
   if kpress then reply chr(kpress);
wend
```

For related information, see the inkey and nextline functions.

Example

```
nchr = nextchar : if null(nchr) then
   gosub a_label
```

In this example, the script tests whether the next character is a blank; if it is, control is passed to the subroutine a_label.

nextline (statement)

Use the nextline statement to get a line of characters from the communications port.

Format

```
nextline <string> [, <time_expr> [, <maxsize>]]
```

nextline accumulates the characters, delimited by carriage returns, that arrive at the communications port and returns them in the variable *string*.

If a carriage return has not been received since the last nextline, the program accumulates characters until a carriage return is encountered, the amount of time specified in *time_expr* is reached, or *maxsize* characters have accumulated. When one of these conditions is met, nextline returns the resulting string and processing continues. If no characters have been received, nextline returns a null string.

time_expr, which can be an integer or a real (floating point) number, is the amount of time, in seconds, to wait for the next carriage return or the next character. If *time_expr* is reached between the receipt of characters, the characters accumulated to that point are returned and script execution continues. You can use the timeout system variable to determine if the value in *time_expr* was exceeded.

time_expr can be any time expression. If *time_expr* is not specified, nextline waits forever to accumulate the number of characters specified by *maxsize* or until a carriage return is received.

maxsize is the number of bytes to accumulate before continuing if a carriage return is not encountered. The default, and maximum, is 255 bytes. A line feed following a carriage return is ignored.

For related information, see the nextchar and nextline functions and the timeout system variable.

Examples

```
nextline new_string
```

In this example, `nextline` waits for characters to come in from the port and stores them in the script's `new_string` variable.

```
nextline big_string, 5.5, 100
if timeout then bye
```

In this example, `nextline` waits for up to 5.5 seconds for as many as 100 characters or a carriage return. The `nextline` statement terminates if the specified conditions are not met within the specified 5.5-second time period. The `timeout` system variable is used to determine whether or not `nextline` timed out.

nextline (function)

Use the nextline function to return a line of characters from the communications port.

Format

```
x$ = nextline[(<delay> [, <maxsize>])]
```

nextline looks for the receipt of a carriage return and then returns the string of characters that have accumulated at the communications port.

If a carriage return has not been received since the last nextline, the characters accumulate until a carriage return is encountered, the amount of time specified in *delay* is reached, or *maxsize* characters have accumulated. The resulting string is then returned and processing continues. If no characters have been received, a null string is returned.

delay is the amount of time to wait for the next carriage return or the next character. If *delay* is reached between the receipt of characters, the characters accumulated to that point are returned and the script continues executing.

The time specified in *delay* is expressed in seconds and can be an integer or real (floating point) number. The default is forever.

maxsize is the number of bytes to accumulate before continuing if a carriage return is not encountered. The default is 255 bytes.

A line feed following a carriage return is ignored.

Versions of Crosstalk for Windows older than 2.0 do not support this function.

For related information, see the nextchar function, the timeout system variable, and the nextline statement.

Examples

```
new_string = nextline
```

In this example, `nextline` waits for characters to come in from the port and stores them in the script's `new_string` variable.

```
big_string = big_string + nextline(15, 1024)
if timeout then bye
```

In this example, `nextline` waits for up to 15 seconds between characters for as many as 1,024 characters or a carriage return. The `nextline` function terminates if a carriage return is received, 1,024 characters are received, or 15 seconds elapses between characters. The characters are accumulated in the variable `big_string`.

null (function)

Use null to determine if a string is null.

Format

```
x = null(<string>)
```

null can be thought of as a simplified version of the length function. length returns the length of a string, but null indicates whether *string* is a null string. Null strings have no length or contents.

null returns true if *string* is null; otherwise, it returns false.

The following code fragments can be thought of as having equivalent meanings when testing the string a_string:

```
if null(a_string) then ...
if length(a_string) = 0 then ...
```

or

```
if length(a_string) then ...
if not null(a_string) then ...
if length(a_string) > 0 then ...
```

For related information, see the length function.

Example

```
print null("Fido"), null("")
```

In this example, the null function displays false for "Fido" and true for "".

number (module variable)

Use number to read or set the phone number for the current session.

Format

number = <*string*>

number sets and reads the phone number associated with the current session. The phone number is limited to 80 characters.

You can specify multiple telephone numbers by separating them with a semicolon. All numbers are dialed until a connection is made. For example, if number is set to the value 1234567;1231111, Crosstalk dials the first number, and if no connection is made, attempts to make a connection using the second number, and so on. If no connection is made, the process is repeated, starting again with the first number, and continues until the numbers have been redialed redialcount times.

For related information, see the redialcount module variable.

Examples

```
number = "5551212"
```

In this example, number is set to 5551212.

```
if number = "5551212" then ...
```

In this example, some action is taken if number is 5551212.

octal (function)

Us `octal` to return a number as a string in octal format.

Format

```
x$ = octal(<integer>)
```

`octal` returns a string containing the octal (base 8) representation of *integer*. The string is 6 or 11 bytes long, depending on the value of *integer*. Table 6-17 shows possible integer ranges and the corresponding byte length.

Table 6-17. Integer ranges for the octal function

Integer ranges	Byte length
0 to 65,535	6
65,536 to 2,147,483,647	11

Example

```
print octal(32767)
```

This example show how to print the octal equivalent of 32,767 decimal.

off (constant)

Use `off` to set a variable to logical false.

Format

```
x = off
```

`off` is always logical false. `off`, like its complement `on`, exists as a way to set variables on and off.

For related information, see the `on`, `false`, and `true` constants.

Example

```
echo = off
```

In this example, `echo` is set to off.

on (constant)

Use on to set a variable to logical true.

Format

```
x = on
```

on is always logical true. on, like its complement off, exists as a way to set variables on and off.

For related information, see the off, false, and true constants.

Example

```
echo = on
```

In this example, the variable echo is set to on.

online (function)

Use `online` to determine if a connection is successful.

Format

```
x = online
```

`online` returns `true` or `false` indicating whether the session is on line to another computer. Some script statements and functions (`reply`, for example) are inappropriate unless you are on line when they are executed. You can use `online` to control program flow.

Examples

```
while online ...
```

In this example, the script performs some task while on line to the host.

```
if not online then call session_name
```

In this example, the script starts the session contained in `session_name` if the session is not on line.

ontime (function)

Use `ontime` to return the number of ticks (one tick is one tenth of a second) this session has been on line.

Format

```
x = ontime
```

You can use `ontime` to call accounting routines, random number routines, and the like.

`ontime` is set to zero when a connection is established and stops counting when the session is disconnected.

Examples

```
print ontime
```

In this example, the script displays the value in `ontime`.

```
if ontime/600 > 30 then ...
```

In this example, the script tests the result of a mathematical computation and takes some action if the result is true.

open (statement)

Use open to open a disk file.

Format

```
open <mode> <filename> as #<filenum>
```

<*mode*> is one of the following:

```
{random | input | output | append}
```

Before a script can read from or write to a file, the file must be opened. open opens *filename* using *filenum* for the activities allowed by *mode*. The *mode* options are described in Table 6-18.

Table 6-18. Mode options for the open statement

Option	Description
random	Allows input and output to the file at any location using seek, get, put, and loc. If the file does not exist, it is created.
input	Allows read-only sequential access of an existing file using read for comma-delimited ASCII records and read line for lines of text. If the file does not exist, a run-time error occurs.
output	Allows write-only sequential access to a newly created file using write for comma-delimited ASCII records and write line for lines of text. If the file exists, it is deleted and a new one is created.
append	Allows write-only sequential access to a file using write for comma-delimited ASCII records and write line for lines of text. If the file exists, the new data is appended to the end of it; otherwise, a new file is created.

filename can be any legal unambiguous file specification; drive specifiers and paths are allowed, but wild cards are not.

filenum must be in the range `1 <= filenum <= 8`. For maximum script flexibility, use the `freefile` function to get the number of an unused *filenum*.

You can open a file in only one mode at a time.

For related information, see the following:

`freefile` function
`get` statement
`loc` function
`put` statement
`read` statement
`read line` statement
`seek` statement
`write` statement
`write line` statement

Examples

```
open random "PATCH.DAT" as #1
```

In this example, the script opens PATCH.DAT in `random` mode with a file number of 1.

```
fileno1 = freefile
open input some_file as #fileno1
```

In this example, the `freefile` number is assigned to `fileno1`, and then the file in `some_file` is opened for input with the file number stored in `fileno1`.

pack (function)

Use pack to return a condensed string.

Format

```
x$ = pack(<string> [, <wild> [, <integer>]])
```

pack returns *string* with duplicate occurrences of the characters in *wild* compressed according to the value of *integer*. *integer* defaults to zero; *wild* defaults to a space.

integer specifies how consecutive characters in *string* are treated. The following *integer* values are valid:

0 All consecutive characters in *string* are compressed to a single occurrence of the first character that appears.

1 Only identical consecutive characters in *string* are compressed.

For example, pack("aabcccdd", "abc", x) returns the following values depending on the value of x:

```
if x = 0, pack returns "add"
if x = 1, pack returns "abcdd"
```

Example

```
pack("HELLO WORLD!", "L", 1)
```

In this example, "HELO WORLD!" is returned because the two L's in HELLO are compressed to one L.

pad (function)

Use pad to return a string padded with spaces, zeros, or other characters.

Format

```
x$ = pad(<orig_str, len_int [, pad_str ...
    [, where_int]]>)
```

pad replaces a host of other functions in conventional programming languages. It can expand, truncate, or center *orig_str* to length *len_int* by adding multiple occurrences of *pad_str* on one or both sides as directed by *where_int*.

pad is essentially the opposite of the strip function, which removes certain characters from a string.

The default for *pad_str* is a space, and the default for *where_int* is 1 (one). This places the padding on the right side of the new string.

where_int has the following meanings:

1 Pads on the right side.
2 Pads on the left side.
3 Pads on both sides, centering *orig_str* in a field *len_int* characters long.

If *len_int* is shorter than length(*orig_str*), *orig_str* is returned, truncated to *len_int* characters with the truncation occurring on the right side of the string.

Examples

```
print pad("Hi", 6); pad("Hi", 6, "-"); ...
   pad("Hi", 4, "+", 2)
```

In this example, the first `pad` function adds 4 spaces to the right of `"Hi"` to expand the string to 6 characters. The second `pad` function adds 4 dashes to the right of `"Hi"` to expand the string to 6 characters. The third `pad` function adds 2 plus signs to the left of `"Hi"` to expand the string to 4 characters. The result is displayed on the screen.

```
cntrd_string = pad("Hello!", 78, "*", 3)
```

In this example, the `pad` function centers `"Hello!"` between two sets of 36 asterisks and returns the result in `cntrd_string`.

password (system variable)

Use password to read or set a password string for the current session.

Format

```
password = <string>
```

password sets and reads the password associated with the current session. The password is limited to 40 characters.

Note: To set the equivalent parameter using your Crosstalk application, choose Session from the Settings pull-down. Then choose the General icon and modify the Password parameter. ■

Examples

```
password = "PRIVATE"
```

This example shows how to set the password.

```
print password
```

This example shows how to print the password.

```
reply password
```

This example shows how to send the password to the host.

patience (module variable)

Use patience to control the amount of time to wait for an answer.

Format

patience = <*integer*>

patience controls the length of time Crosstalk waits for the host to answer. If the appropriate carrier tone is not reported by the modem in *integer* seconds after the dialing process was initiated, Crosstalk hangs up the telephone. The maximum value for *integer* is 999.

redialcount, patience, and redialwait control the process of dialing, waiting for carrier, and waiting to redial. Redialing is independent of and transparent to scripts.

In the United States and other countries with similar telephone systems, a patience setting of 30 will generally prove reliable, striking a good balance between waiting too long and hanging up too soon. If you are calling internationally, are using private telephone network services, or are in a location served by some types of older telephone equipment, you may need to set patience to 45 seconds or 60 seconds.

Most modems have a similar setting, and default to a 30-second wait period. patience controls the amount of time Crosstalk waits for a call, not the amount of time the modem will wait. Check your modem documentation for information on modifying the wait-for-carrier time.

Government or telephone authority regulations may specify the minimum or maximum amount of time that you can allow a telephone to ring. It is your responsibility to adhere to the appropriate regulations concerning telephone use in your locality.

For related information, see the redialcount and redialwait module variables.

Example

```
patience = 30
```

In this example, patience is set to a 30-second wait time.

perform (statement)

Use `perform` to call a procedure.

Format

`perform <procedurename> [<arglist>]`

`perform` is an alternate method of calling a procedure. It is like a forward declaration and a call, all in one. Its use is optional. Use it to call procedures when they are located near the end of the script.

procedurename is the name of the procedure that is called. *arglist* is a list of arguments that can be passed to the procedure. *arglist* must contain the same number and types of arguments and in the same order as specified in the procedure declaration. Be sure to separate the arguments with commas.

Versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information, see the `proc ... endproc` procedure declaration.

Example

```
perform some_proc
```

In this example, the procedure identified by `some_proc` is called.

pop (statement)

Use pop to remove a return address from the gosub return stack.

Format

```
pop
```

You can use pop in a subroutine to alter the flow of control. pop removes the top address from the gosub return stack so that a subsequent return statement returns control to the previous gosub rather than the calling gosub.

When you use the pop statement, the logic of your script becomes somewhat obscure; therefore, use the statement only on those occasions where it cannot be avoided.

If the return stack is empty when the pop statement is encountered, an "underflow" error occurs.

For related information, see the gosub ... return statements.

Example

```
pop
```

press (statement)

Use press to send a series of keystrokes to the terminal emulator.

Format

```
press [<string> [, <string>] ... ] [;]
```

Normally, press sends special keys that are dependent on the type of terminal in use. For example, the following statement simulates the pressing of the HOME key.

```
press "<Home>"
```

If you are using a VT™ 100 terminal, the VT100 codes for the HOME key are sent.

string is a string expression containing the keys to be sent. To suppress the trailing carriage return, use a semicolon at the end of the statement.

Note: Enclose special key names in angle brackets: "<F1>" rather than "F1." Characters in the string that are not enclosed in angle brackets are sent as plain text characters. If you need to send one of the unnamed keys such as Ctrl-7, place the key number inside the angle brackets. (See the inkey function earlier in this chapter for a list of key numbers. You can access a key map for the terminal you are using from the Crosstalk Keyboard Editor.) ■

The difference between press and reply is subtle. reply always sends its output directly to the communications device while press passes its output through the terminal emulator, just as if you had pressed a key on the keyboard. reply does not honor any special key codes that are part of the terminal emulator; press does honor such key codes.

This statement is valid only when you are on line.

Versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information, see the reply statement.

Examples

```
keys_out = "<up><left>" : press keys_out ;
```

In this example, the special keys `up` and `left` are assigned to the variable `keys_out`, which is sent using the `press` statement.

```
press "Crosstalk";
```

In this example, the script sends the string Crosstalk without a trailing carriage return.

```
press "<8>" ;
```

In this example, the script sends a backspace, which is represented by the number 8.

print (statement)

Use `print` to display text in a window.

Format

`print [<item>] [{ , | ; } [<item>]] ... [ ; ]`

<item> is one of the following:

`{<expression> | at <row>, <col>}`

The keyword `at` specifies a position in the window; if it is omitted, printing begins at the current cursor position.

The *items* can be any expression or list of expressions, including integers, strings, and quoted text, separated by semicolons or commas. If the items in the list are separated by semicolons, they are printed with no space between them. If the items are separated by commas, they are printed at the next tab position. If no expression is provided, a blank line is printed.

A trailing semicolon at the end of the `print` statement causes the item to be printed without a carriage return. This is useful when you want to print something immediately after the statement on the same line, or when printing on the last line of a window.

`print` can be abbreviated as "?".

Note: If a script sets `display` to `off`, `print` statements do not display text in the window. ■

For related information, see the `display` statement.

Examples

```
print "Current protocol is " ; protocol
```

In this example, the script prints the text "Current protocol is " followed by the name of the selected protocol.

```
print "This is all printed on the ";
print "same line."
```

In this example, the script prints the text on a single line.

```
print date , time(-1)
```

In this example, the script prints the date and the current time, with the time starting at the next tab stop.

printer (system variable)

Use `printer` to send screen output to the printer.

Format

```
printer = {on | off}
```

`printer` turns printing `on` or `off`. When `printer` is `on`, Crosstalk sends the stream of characters coming from the communications port to the system printer.

Note that Crosstalk's VT102, VT52, and IBM® 3101 emulations have the ability to turn the printer on or off upon receipt of a command sequence from a host. In this case, the printer is controlled automatically and does not need to be turned on by a script or the user.

Example

```
printer = off
```

This example shows how to turn printing off.

proc ... endproc (procedure declaration)

Use `proc ... endproc` to define and name a procedure.

Format

```
proc <name> [takes [<type>] <argument>
     [, [<type>] <argument>]...]
     ...
     ...
endproc
```

A procedure is a group of statements that can be predefined in a script and later referred to by name. Procedures can take a number of arguments; the arguments are optional. If arguments are included, you must use the same number and type of arguments in both the procedure and the statement that calls the procedure. The arguments are assumed to be strings unless otherwise specified.

name is the name given to the procedure. It must be a unique name.

`takes` is optional and describes a list of arguments that are passed to the procedure.

`endproc` terminates the procedure. If you want to leave a procedure before the `endproc`, use the `exit` statement to return control to the calling routine.

Any variable declared within a procedure is local to the procedure. The procedure can reference variables that are outside the procedure, but not the other way around.

Procedures can contain labels, and the labels can be the target of `gosub ... return` and `goto` statements, but such activity must be wholly contained within the procedure. If you reference a label inside a procedure from outside the procedure, an error occurs.

You can nest procedures at the execution level; that is, one procedure can call another. You must not nest procedures at the definition level; one procedure definition cannot contain another procedure definition.

You can use forward declarations to declare procedures whose definition occurs later in the script. The syntax of a forward procedure declaration is the same as the first line of a procedure definition, with the addition of the `forward` keyword.

Forward declarations are useful if you want to place your procedures near the end of your script. A procedure must be declared before you can call it; the forward declaration provides the means to declare a procedure and later define what the procedure is to perform.

The following format is used for a forward declaration:

```
proc <name> [takes <arglist>] forward
```

You can also use the `perform` statement to call a procedure that is not yet declared.

Procedures can be in separate files. To include an external procedure in a script, use the `include` compiler directive.

Versions of Crosstalk for Windows older than 2.0 do not support the procedure declaration.

For related information, see the `func ... endfunc` function declaration; the `exit`, `gosub ... return`, `goto`, and `perform` statements; and the `include` compiler directive.

Examples

```
proc logon takes string username, ...
    string logon_password
    watch for
        "Enter user ID:"  : reply username
        "Enter password:" : reply logon_password
        key 27            : exit
    endwatch
endproc
```

In this example, `username` and `logon_password` are the procedure arguments. The values of `username` and `logon_password` are passed to the procedure when it is called. The procedure watches for the appropriate prompts from the host and responds with one or the other of the arguments. If the ESC key is received, the procedure exits to the calling routine.

```
proc logon takes string username, string ...
   logon_password forward

logon "John", "secret"

proc logon takes string username, ...
   string logon_password
   watch for
      "Enter user ID:"  : reply username
      "Enter password:" : reply logon_password
      key 27            : exit
   endwatch
endproc
```

In this example, the procedure logon is declared as a forward declaration. Then it is called.

Note: For ease of programming, you do not have to supply the parameters in the actual procedure definition if you use a forward declaration. For instance, the foregoing example can also be written as follows:

```
proc logon takes string username, ...
   string logon_password forward

logon "John", "secret"

proc logon
   watch for
      "Enter user ID:"  : reply username
      "Enter password:" : reply logon_password
      key 27            : exit
   endwatch
endproc ■
```

protocol (system variable)

Use `protocol` to set or read the protocol setting.

Format

`protocol = <string>`

`protocol` checks or changes the protocol to use for file transfers.

`string` can be one of the file transfer protocols listed in Table 6-19.

Table 6-19. File transfer protocols

Protocol name	Sub-models (use the `protomodel` variable)	Functionality

`DCAXYMDM` * or `DCA XYMODEM Tool`†	`XMODEM`
	`XMODEM/CRC`
	`XMODEM/1K`
	`XMODEM/G`
	`YMODEM/BATCH`
	`YMODEM/G`
		Loads the XMODEM/ YMODEM tool. The default is XMODEM/CRC.

`DCACSERV`* or `DCA CServeB Tool`†	(None)	Loads the CompuServe B file transfer tool.
`DCAZMDM` * or `DCA ZMODEM Tool`†	(None)	Loads the ZMODEM tool.
`DCAKERMT`* or `DCA KERMIT Tool`†	(None)	Loads the Kermit tool.

* Windows environment
† Macintosh environment

continued

Table 6-19. File transfer protocols (cont.)

Protocol name	Sub-models (use the protomodel variable)	Functionality
DCAIND* or DCA IND$FILE Tool†	(None)	Loads the IND$FILE tool.
DCAXTALK*	(None)	Loads the Crosstalk XVI tool.
DCADART* or DCA DART Tool†	(None)	Loads the Crosstalk DART tool.

* Windows environment
† Macintosh environment

Note: To set the equivalent parameter using your Crosstalk application, choose File Transfer from the Settings pull-down. ■

For related information, see the `assume` statement and the `device` and `terminal` system variables.

Examples

```
assume protocol "DCAXYMDM"
protocol = "DCAXYMDM"
protomodel = "YMODEM/BATCH"
```

In this example, the `DCAXYMDM` file transfer tool is loaded with the `YMODEM/BATCH` sub-model specified.

```
print protocol
```

In this example, the script prints the current protocol selection.

```
if protocol = "DCAXYMDM" then ...
```

In this example, the script takes some action if the protocol selected is `DCAXYMDM`.

put (statement)

Use put to write characters to a random file.

Format

```
put [#<filenum>, ] <string>
```

put writes *string* to the random file specified by *filenum*. length(*string*) is the number of bytes written to the file. *filenum* must be an open random file number.

If the end-of-file marker is reached during the write, the file is extended.

Each put advances the file I/O pointer by length(*string*) positions. The put statement does not pad *string* to a particular length (to pad the string, you must use the pad function), nor does it add quotation marks, carriage returns, or end-of-file markers.

You must open the file in random mode.

For related information, see the defoutput system variable, the pad function, and the open and seek statements.

Examples

```
put #1, some_string
```

In this example, the script writes some_string to a file with a file number of 1.

```
put #fileno1, pad(rec, rec_len)
```

In this example, rec is padded on the right with spaces to expand the string to rec_length characters, and then rec is put to the file designated by fileno1.

quit (statement)

Use `quit` to close a session window.

Format

```
quit
```

This statement ends a Crosstalk session. Unlike the `terminate` statement, `quit` does not end the Crosstalk application, even if you use the statement to end the last or only active session.

Note: To perform the same function using your Crosstalk application, choose Close from the session window's System menu. ■

For related information, see the `terminate` statement.

Example

```
quit
```

quote (function)

Use quote to return a string enclosed in quotation marks.

Format

```
x$ = quote(<string>)
```

quote analyzes *string* and returns it enclosed in quotation marks to make it compatible with the type of comma-delimited ASCII sequential file input/output used by many applications.

quote encloses any string that contains a comma in double (") quotation marks.

string cannot contain both single and double quotation marks.

Example

```
print quote("Hello, world!")
```

In this example, the phrase Hello, world! is enclosed in double quotation marks when it is displayed on the screen.

read (statement)

Use read to read data from a sequential disk file.

Format

```
read [#<filenum>, ] <string_var_list>
```

The read statement operates only on files opened in input mode.

filenum must be an open input file number. If *filenum* is not supplied, the default input file number, which is stored in definput, is assumed.

The read statement reads lines containing comma-delimited fields of ASCII data. Each read puts fields into the members of *string_var_list* until either all of the members have had values assigned, or the end-of-file marker is reached. Quotation marks are automatically stripped. When end-of-line is reached, it is treated as a comma (delimiter).

To use the read statement, you must have previously defined all members of *string_var_list*.

For related information, see the definput system variable and the open and read line statements.

Example

```
read #fileno, alpha, beta, gamma
```

In this example, the read statement uses file number #fileno to read fields of ASCII data into the variables alpha, beta, and gamma.

read line (statement)

Use `read line` to read data from a sequential disk file.

Format

```
read line [#<filenum>, ] <string_var>
```

Like the `read` statement, the `read line` statement operates only on files opened in `input` mode.

filenum must be an open input file number. If *filenum* is not supplied, the default input file number, which is stored in `definput`, is assumed.

The `read line` statement reads lines of text from files. Each `read line` puts in *string_var* all the text read, up to the next carriage-return/line-feed (CRLF) character or a maximum of 255 characters, whichever comes first. If the end-of-file marker has already been reached, *string_var* is null.

To use the `read line` statement, you must have previously declared *string_var*.

For related information, see the `definput` system variable and the `open` and `read` statements.

Example

```
read line #1, some_text
```

In this example, the `read line` statement uses the file number #1 to read a line of text into the variable `some_text`.

receive (statement)

Use receive to receive a file from another computer.

Format

receive <*filename*>

receive tells Crosstalk to begin receiving a file or group of files from the computer at the other end of the connection. *filename* is the name of the file to be received. The file is saved using the same name and is placed in the directory defined for transfers. (See the dirfil and downloaddir system variables earlier in this chapter for details.)

The way receive works depends on the protocol you use. For example, some protocols such as DART understand how to request information from the host while other protocols such as XMODEM require user intervention to request data.

Note that if the selected protocol is CompuServe B, *filename* is not required.

An error occurs if the statement is executed while you are not on line.

Note: To start receiving files using your Crosstalk application, choose File Transfer from the Action pull-down and then choose Receive Files(s). ■

For related information, see the send statement and the dirfil and downloaddir system variables.

Examples

```
receive
```

In this example, receive requests a file using the CompuServe B protocol.

```
receive "B:ERNIE"
```

In this example, `receive` requests a file called ERNIE from the remote system's drive B.

```
receive fname
```

In this example, `receive` requests the file with the name assigned to the `fname` variable.

```
receive "ERNIE"
```

In this example, `receive` requests a file using the name ERNIE.

redialcount (module variable)

Use `redialcount` to control the number of times a telephone number is redialed.

Format

`redialcount = <integer>`

`redialcount` controls the number of times a busy or unanswered telephone number is redialed. The number is attempted *integer* plus one time before dialing is discontinued. The maximum number for *integer* is 99. A `redialcount` of zero means the number is dialed one time. Redialing is independent of and transparent to scripts.

Government or telephone authority regulations may specify the maximum number of times an automated device can dial a single telephone number. In the United States, the Federal Communications Commission (FCC) has set this maximum at 15. The limit in Canada is 10. It is your responsibility to adhere to the appropriate regulations concerning telephone use in your locality.

For related information, see the `redialwait` and `patience` module variables.

Example

```
redialcount = 9
```

In this example, dialing is attempted 10 times.

redialwait (module variable)

Use `redialwait` to control the amount of time between redials.

Format

`redialwait = <integer>`

`redialwait` controls the length of time Crosstalk waits before attempting to redial a busy or unanswered telephone number. If the number dialed is busy or goes unanswered, Crosstalk waits *integer* seconds before trying again unless the value of the `redialcount` module variable has been reached. The maximum number for *integer* is 99. Redialing is independent of and transparent to scripts.

Check your government or telephone authority regulations to learn if there is a minimum amount of time that can elapse between consecutive attempts to connect with a single telephone number.

For related information, see the `redialcount` and `patience` module variables.

Example

```
redialwait = 30
```

In this example, the script waits 30 seconds before attempting to redial a phone number.

rename (statement)

Use rename to rename a file.

Format

```
rename [some] <oldname>, <newname>
```

This statement renames a file. *oldname* must be the name of an existing file and can contain wild cards. If some is specified, the user is prompted to verify each file before it is renamed.

Examples

```
rename "TEST.XWS", "MAIL.XWS"
```

In this example, the script renames the existing file TEST.XWS to MAIL.XWS.

```
rename FNAME1, FNAME2
```

In this example, the script renames the file in the FNAME1 variable to the name in the FNAME2 variable.

repeat ... until (statements)

Use repeat ... until to repeat a statement or series of statements until a given condition becomes true.

Format

```
repeat
   ...
   ...
   ...
until <expression>
```

repeat lets you repeat a group of statements until some condition occurs. until specifies the condition that terminates the repeat condition. *expression* can be any boolean, numeric, or string expression.

The loop is executed once before *expression* is checked. If *expression* is false, the loop is repeated until expression is true.

The repeat/until construct is a good alternative to the while/wend construct in those instances where a loop must be executed at least once before its terminating condition is tested.

For related information, see the while ... wend statements.

Examples

```
x = 0
repeat
   x = x + 1
   print x
until x = 100
```

In this example, the script prints numbers from 1 to 100.

```
string guess
print "Guess how to get out of here:"
repeat
   input guess
until guess = "Good Bye!"
```

This example shows how a script can prompt the user to enter a string and repeat the prompt until the correct string (`Good Bye!`) is entered.

reply (statement)

Use `reply` to send a string of text to the communications device.

Format

```
reply [<string> [, <string>] ... ] [;]
```

`reply` sends one or more strings of text directly to the communications device. *`string`* is a string expression containing the text to be transmitted.

`reply` sends a carriage return after it sends *`string`*. To suppress this action, use a semicolon at the end of the statement. If you use the statement without an argument, it sends only a carriage return.

Use this statement only when you are on line.

For related information, see the `press` statement.

Examples

```
reply "Hello!"
```

In this example, the script sends `Hello!`

```
reply userid + " " + password
```
or
```
reply userid, " ", password
```
or
```
reply userid;
reply " ";
reply password
```

In this example, the script sends the user ID, a space, and the password.

```
reply chr(3);
```

In this example, the script sends a ^C to the host.

request (statement)

The `request` statement, which is a synonym for the `receive` statement, is supported only for backward compatibility. Refer to `receive` earlier in this chapter.

restore (statement)

Use `restore` to restore the Crosstalk application window to its previous size.

Format

```
restore
```

The `restore` statement is functionally equivalent to choosing the Restore option from the application window's Control menu.

Crosstalk Mark 4 does not support this statement.

For related information, see the `maximize`, `minimize`, `move`, and `size` statements.

Example

```
restore
```

return (statement)

Use `return` to exit a function or to return from a subroutine.

Format

`return [<expression>]`

When the `return` statement is used to exit a function, it returns a value. *`expression`* is the return value.

When `return` is used in a subroutine, the statement does not return a value.

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 support only the return from a subroutine.

For related information, see the `func ... endfunc` function declaration and the `gosub ... return` statements.

Examples

```
func calc_largest (integer num1,  ...
     integer num2) returns integer
   if num1 > num2 then return num1
   else return num2
endfunc
```

In this example, the function compares 2 numbers to determine which is larger and returns that number.

```
integer i

gosub count_to_10
end

label count_to_10
   for i = 1 to 10
      print i
   next
return
```

In this example, the script calls a subroutine to display the numbers 1 to 10. Note that the `return` statement does not return a value in this example.

rewind (statement)

Use `rewind` to move the next-character pointer backwards in the capture buffer.

Format

`rewind <integer>`

Crosstalk maintains a pointer to the position in the capture buffer where the next character should be stored. `rewind` provides the means to move this pointer backwards *integer* characters if you want to overwrite information in the buffer.

This statement is effectively the opposite of the `add` statement, which lets you add strings of data to the capture buffer.

Versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information, see the `add` statement.

Example

```
rewind 8
```

In this example, the pointer in the capture buffer is moved back 8 characters.

right (function)

Use `right` to return the right portion of a string.

Format

```
x$ = right(<string [, integer]>)
```

`right` returns the rightmost *integer* characters in *string*. If *integer* is not specified, the last character in *string* is returned. If *integer* is greater than the length of *string*, *string* is returned.

Examples

```
dog_name = right("Hey, Fido", 4)
```

In this example, `right` returns `"Fido"` in dog_name.

```
print right(long_string, 78)
```

In this example, the last 78 characters in `long_string` are printed to the screen.

rmdir (statement)

Use rmdir to remove a subdirectory.

Format

```
rmdir <directory>
```

directory must be a string expression containing a valid directory name. If the directory name exists and contains no files or directories, it is removed. If it does not exist or if it contains files or subdirectories, an error occurs.

Note: You can also use the abbreviation rd for this statement. ■

Examples

```
rmdir "C:\XTALK\TMP"
```

In this example, the rmdir statement removes the TMP subdirectory.

```
rmdir some_dirname
```

In this example, rmdir removes the directory contained in some_dirname.

run (statement)

Use `run` to run another application.

Format

`run <pathname>`

This statement starts another application. Crosstalk and the new application run concurrently.

In a Windows environment, if the application name is supplied without a path, the application program file must reside in the DOS path. If the application resides elsewhere, it must be preceded by the path to the program.

Examples

```
run "NOTEPAD.EXE"
```

In this example, the application NOTEPAD.EXE is run.

```
run "D:\APPS\CLOCK.EXE"
```

In this example, the application CLOCK.EXE, which is located in the APPS directory on drive D, is run.

save (statement)

Use save to save session parameters.

Format

save [<*name*>]

name is optional. If provided, it must be a valid file name for your operating environment. If *name* is not provided, the current name is used.

This statement saves all of the information associated with the session currently in use, including the phone number and description. If the session is untitled when this statement is executed, Crosstalk creates a session profile with the current settings and names it TEMP.XWP for the Windows environment or Temp Session for the Macintosh environment.

Examples

```
save
```

In this example, the script saves the session settings using the current name.

```
save "Source"
```

In this example, the script saves the session settings using the name provided.

script (system variable)

Use `script` to specify the name of the logon script file used by the current session.

Format

`script = <filename>`

`script` specifies the name of the script file to use for the current session. *`filename`* must be a valid file name for your operating environment.

Examples

```
script = "CSERVE"
```

In this example, the session script is set to CSERVE.

```
if script = "MCIMAIL" then ...
```

In this example, some action is taken if the script for the session is named MCIMAIL.

scriptdesc (compiler directive)

Use `scriptdesc` to specify a description for a script.

Format

```
scriptdesc <string>
```

The `scriptdesc` compiler directive defines descriptive text for a script. *`string`* can be up to 40 characters in length.

Win

When the script is added to the Script pull-down and to the Open dialog box, the `scriptdesc` text appears next to the appropriate script name. ■

Versions of Crosstalk for Windows older than 2.0 do not support this directive.

Example

```
scriptdesc "Login script for the VAX system"
```

In this example, `scriptdesc` is set to the specified string.

secno (function)

Use secno to return the number of seconds since midnight.

Format

```
x = secno[(<hh>, <mm>, <ss>)]
```

secno returns the number of seconds since midnight.

You can get the number of seconds that have elapsed since midnight for any given time by passing the hours, minutes, and seconds of that time as *hh*, *mm*, and *ss*.

Examples

```
print secno
```

In this example, the elapsed seconds since midnight are printed.

```
print secno(14, 2, 31)
```

In this example the script prints the elapsed seconds since midnight for the time 2:02:31 PM.

seek (statement)

Use seek to move a random file input/output pointer.

Format

```
seek [#<filenum>, ] <integer>
```

seek moves a random file input/output pointer to character position *integer*. The next file get or put action commences at that point. (Note that the first byte in a file is character position zero.) *integer* is the number of bytes from the beginning of the file, not the current location. (See the loc function earlier in this chapter for more information.)

seek does not move the pointer beyond the end-of-file marker.

Each get or put advances the input/output pointer by the number of bytes read or written. If the records in a random file are of fixed length and each get reads one record, reading the file backwards requires that after each get you must seek backwards two records.

You must open the file in random mode to use this statement.

For related information, see the get, open, and put statements and the loc function.

Examples

```
seek #1, 0
```

In this example, the pointer is positioned at the beginning of the file.

```
seek #1, rec_len * rec_num
```

In this example, seek moves the I/O pointer to the position that results from multiplying the record length by the record number.

send (statement)

Use send to transfer a file or group of files to another computer.

Format

```
send <filename>
```

send initiates a file transfer to another computer. *filename* is the name of the file to send, and can be a full path name.

The operation of this command is dependent on the file transfer protocol in use. If you are using the Crosstalk, DART, YMODEM/Batch, ZMODEM, or Kermit protocols, the send statement can send multiple files. If you are sending multiple files, you can specify a wild-card file name in *filename*.

The XMODEM and XMODEM/1k protocols do not allow you to send more than one file at a time.

This statement is valid only when you are on line.

Note: To send a file using your Crosstalk application, choose File Transfer from the Action pull-down and then choose Send File(s). ■

For related information, see the receive statement.

Examples

```
send "B:ERNIE"
```

In this example, the send statement sends the file ERNIE from drive B on the sending computer to the other computer.

```
send some_fname
```

In this example, the send statement sends the file assigned to some_fname.

sendbreak (statement)

Use `sendbreak` to send a break signal to the host.

Format

```
sendbreak
```

This statement sends a break signal to the host. Break signals are often interpreted by host systems as a "cancel" signal, and they usually stop some action.

The length of the break signal is controlled either by the Break Length setting in the Connection Settings dialog box, which you can access by choosing Connection from the Settings pull-down, or by the `breaklen` module variable setting.

This statement is valid only when you are on line.

For related information, see the `breaklen` module variable.

Example

```
sendbreak
```

session (function)

Use `session` to find out the current session number.

Format

```
x = session
```

The `session` function returns the session number of the current session, which may or may not be the active session. The active session is defined as the session that is currently using the keyboard or is waiting for keyboard input. The current session is the one in which the script is running.

To determine if the script currently running is the active session, test both the `activesession` and the `session` functions.

Versions of Crosstalk for Windows older than 2.0 do not support this function.

For related information, see the `activesession` function.

Example

```
if activesession = session then
   reply "The current session is the " + ...
      "active session."
```

In this example, the `session` and `activesession` functions are compared to find out if the active session is the current session.

sessname (function)

Use sessname to find out the name of another session.

Format

```
x$ = sessname(<integer>)
```

sessname returns the name of the session represented by *integer*. If there is no session with that number, a null string is returned.

You can use this function to find out what sessions are running concurrently.

Versions of Crosstalk for Windows older than 2.0 do not support this function.

For related information, see the sessno function.

Example

```
print sessname(1), sessno(sessname(1))
```

In this example, the script displays the name and number of the session identified by the integer 1.

sessno (function)

Use sessno to find out the session number of a specified session.

Format

```
x = sessno[(string)]
```

sessno returns the number of the session whose name is *string*. If there is no session with that name, 0 (zero) is returned. If you do not specify an argument, sessno returns the number of open sessions.

As with the sessname function, you can use this function to find out what sessions are running concurrently.

Versions of Crosstalk for Windows older than 2.0 do not support this function.

For related information, see the sessname function.

Example

```
if sessno ("CSERVE") then
   print "A CompuServe session exists."
```

In this example, the script displays a message if one of the currently open sessions is CSERVE.

show (statement)

Use `show` to redisplay a Crosstalk session window.

Format

```
show
```

This command redisplays a Crosstalk session window that was previously reduced to an icon with the `hide` statement.

Example

```
show
```

showallquickpads (statement)

Use showallquickpads to show all of the QuickPads that are loaded for the current session.

Format

```
showallquickpads
```

This statement displays all of the QuickPads that were previously hidden.

Note: The QuickPads for the session must already be loaded using the loadquickpad or loadallquickpads statement. ■

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information see the hideallquickpads, hidequickpad, loadallquickpads, loadquickpad, and showquickpad statements.

Example

```
showallquickpads
```

showquickpad (statement)

Use showquickpad to show the specified QuickPad for the current session.

Format

```
showquickpad <string>
```

This statement displays the QuickPad specified in *string*.

Note: The QuickPad for the session must already be loaded using the loadquickpad or loadallquickpads statement. ■

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information see the hideallquickpads, hidequickpad, loadallquickpads, loadquickpad, and showallquickpads statements.

Example

```
showquickpad "sessA"
```

In this example, the QuickPad identified as sessA is displayed.

Win size (statement)

Use `size` to change the size of the Crosstalk application window.

Format

`size <x>, <y>`

This statement changes the size of the Crosstalk application window. The window can be made larger or smaller than its current size.

x and *y* are the horizontal and vertical size, in pixels.

The `size` statement performs the same function as the Size option (ALT-F8) from the application window's Control Menu.

The range of coordinates is determined by the resolution of the display adapter and monitor in use.

For related information, see the `maximize`, `minimize`, `move`, and `restore` statements.

Example

```
size 200, 350
```

In this example, the application window is resized to be 200 pixels wide and 350 pixels high. ■

slice (function)

Use `slice` to return portions of a string.

Format

```
x$ = slice(<string, integer ...
    [, delin_str [, where_int]]>)
```

`slice` breaks out portions of strings. *`string`* is divided into substrings as delineated by occurrences of *`delin_str`*. *`delin_str`* can specify more than one delimiter (for example, ";:"); it defaults to a space.

The substring in *`integer`* position is returned.

`where_int` specifies where the function is to begin its analysis in *`string`*.

Examples

```
sub_string = slice("alpha beta gamma", 2)
```

In this example, `slice` returns "beta."

```
print slice("alpha, beta, gamma", 2, ",")
```

In this example, "beta" is displayed on the screen.

```
sub_string = slice("alpha, beta gamma.delta", 3,",.")
```

In this example, `slice` returns "delta."

startup (system variable)

Use `startup` to read or set the name of a script to run when Crosstalk is started.

Format

`startup = <string>`

`startup` sets or reads the name of the script you want to run automatically whenever a new session is started. If `startup` is null, no script is run at start-up time. *`string`* must be a valid file name for your operating environment.

Examples

```
startup = "AUTOEXEC"
```

In this example, a script called AUTOEXEC is run when Crosstalk is started.

```
startup = ""
```

In this example, `startup` is null, so no script is run when Crosstalk is started.

str (function)

Use `str` to convert a number to string format.

Format

```
x$ = str(<number>)
```

The `str` function is the opposite of the `val` and `intval` functions in that it converts numbers to strings. *`number`* can be a real (floating point) number or an integer. `str` does not add any leading or trailing spaces.

For related information, see the `intval` and `val` functions.

Examples

```
print 2 : print str(2) : print length(str(2))
```

In this example, the script displays 3 lines. The first line contains the integer 2. The second line contains the string that results from converting integer 2 to a string. The last line contains the length of the string displayed in line 2.

```
reply str(shares_to_buy)
```

In this example, the script sends the string equivalent of `shares_to_buy` to the host.

```
integer counter
string items[10]
for counter = 1 to 10
   items[counter] = "item" + str(counter)
   print items[counter]
next
```

In this example, the script declares `counter` as an integer and `items` as an array of 10 strings. The `for/next` construct is used to display the individual elements in the array.

strip (function)

Use strip to return a string with certain characters removed.

Format

```
x$ = strip(<string [, wild [, where_int]]>)
```

strip removes unwanted characters from strings. It is essentially the opposite of the pad function, which pads a string with spaces, zeros, or other characters.

wild can be either a string of characters you want to remove from *string* or an integer bit-map of the Crosstalk character class(es) containing the characters you want removed. (Refer to the class function earlier in this chapter for additional information.) The default for *wild* is a space.

where_int has the following meanings:

0 Strip out all occurrences in *string* of any character in *wild*. This is the default.

1 Strip from the right side, stopping at the first occurrence of a character not in *wild*.

2 Strip from the left side, stopping at the first occurrence of a character not in *wild*.

3 Strip from both the right and left sides, stopping on each side at the first occurrence of a character not in *wild*.

strip is quite useful in removing "junk" characters from lines read from word-processing text files, for removing leading zeros, and for cleaning up user-entered strings.

For related information, see the class and pad functions.

Examples

```
print strip("0123456", "0", 2)
```

In this example, the script displays "123456."

```
print strip("Sassafras", "as", 0)
```

In this example, the script prints "fr."

```
reply strip(strip(user_resp, junk, 0), " ", 3)
```

In this example, the script first strips out "junk" from `user_resp` and then strips leading and trailing spaces from what remains of `user_resp`. The result is sent to the host.

stroke (function)

Use `stroke` to wait for the next keystroke from the keyboard.

Format

```
x = stroke
```

`stroke` is similar to the `inkey` function, but `stroke` stops the script to wait for a keystroke and returns the value of the keystroke. The value returned is the ASCII value of the key pressed for the printable characters (0–127 decimal) and special keystrokes such as the arrow keys, function keys, and special-purpose keys. (See the `inkey` function earlier in this chapter for a list of appropriate keys and their corresponding numbers.)

Versions of Crosstalk for Windows older than 2.0 do not support this function.

Example

```
print "Press a key to see its value"; : print stroke
```

In this example, the script prints a message followed by the value of the key that was pressed.

subst (function)

Use subst to return a string with certain characters substituted.

Format

```
x$ = subst(<string, old_str, new_str>)
```

For each character in *old_str* that subst finds in *string*, it substitutes the corresponding character in *new_str*.

Example

```
print subst("alpha", "a", "b")
```

In this example, the script prints "blphb."

systime (function)

Use `systime` to return the number of ticks Crosstalk has been active.

Format

```
x = systime
```

`systime` returns the number of ticks the Crosstalk application has been active. One `tick` is one tenth of a second. You can use `systime` in delay loops, random number routines, and the like.

Examples

```
print systime
```

In this example, the value in `systime` is displayed.

```
if systime mod 100 = 0 then ...
```

In this example, the script takes some action if the value of `systime` divided by 100 is zero.

tabex (system variable)

Use `tabex` to control the expansion of tabs to spaces.

Format

```
tabex = {on | off}
```

`tabex` determines whether Crosstalk sends outgoing tab (ASCII decimal 9) characters as spaces during ASCII text uploads.

If `tabex` is `on`, Crosstalk expands a file's tab characters to 8 spaces.

This is most useful when uploading a file containing tab characters to a host computer that does not understand what tab characters are.

Example

```
tabex = off
```

In this example, tab characters are not expanded to spaces.

tabwidth (module variable)

Use `tabwidth` to determine the number of spaces a tab character moves the cursor.

Format

`tabwidth = <integer>`

This variable determines the number of spaces the cursor is moved when the tab character is received. *integer* can be any number from 1 to 80. The default is 8.

Crosstalk Mark 4 does not support this variable.

Example

```
tabwidth = 15
```

In this example, `tabwidth` is set to 15 spaces.

terminal (system variable)

Use `terminal` to read or set the name of the terminal emulation module used by the session.

Format

`terminal = <string>`

`terminal` specifies the name of the terminal emulation to use for the current session. *`string`* can be one of the terminal emulations found in Table 6-20.

Table 6-20. Terminal emulations

Emulation name	Sub-models (use the `termmodel` variable)	Functionality in the tool
`DCADEC`* or `DCA DEC Tool`†	`VT52, VT102, VT220, VT320`	Loads the DEC® tool. The default is VT102.
`DCAANSI`* or `DCA ANSIPC Tool`†	(None)	Loads the ANSI.SYS tool.
`DCAVIDTX`* or `DCA VIDTEX Tool`†	(None)	Loads the CompuServe Vidtex™ tool.
`DCATTY`* or `DCA TTY Tool`†	(None)	Loads the generic TTY tool.

* Windows environment
† Macintosh environment

continued

Table 6-20. Terminal emulations (cont.)

Emulation name	Sub-models (use the `termmodel` variable)	Functionality in the tool
`DCAIBM` * or `DCA IBM3101 Tool`†	(None)	Loads the IBM 3101 tool.
`DCAFTTRM` * or `DCA FTTERM Tool`†	(None)	Loads the IBM FTTERM tool.
`DCAWYSE`* or `DCA WYSE Tool`†	`WYSE 50`, `WYSE 50+`, `WYSE 60`, `ADDS VIEWPOINT`, `HAZELTINE 1500`, `PC-TERM`, `TELEVIDEO 912`, `TELEVIDEO 920`, `TELEVIDEO 925`	Loads the WYSE™ emulation and its sub-emulations. The default is WYSE 60.
`DCAHP700`* or `DCA HP700/94 Tool`†	(None)	Loads the HP® 700/94 tool.

* Windows environment
† Macintosh environment

Note: To set the equivalent parameter using your Crosstalk application, choose Terminal from the Settings pull-down. ■

For related information, see the `assume` statement and the `device` and `protocol` system variables. Refer to your Crosstalk user's guide for more information on terminal emulation.

Examples

```
assume terminal "DCAWYSE"
terminal = "DCAWYSE"
termmodel = "WYSE 50"
```

This example shows how to load the DCAWYSE terminal tool with WYSE 50 emulation.

```
print terminal
```

This example shows how to print the current terminal emulation selection.

```
terminal = "DCAIBM"
```

In this example, terminal is set to IBM 3101 terminal emulation.

```
string term_type
term_type = terminal
if term_type <> "DCAIBM" then
   terminal = "DCAIBM"
```

In this example, the value in terminal is assigned to the string term_type. term_type is then tested to determine if it contains the value DCAIBM. If not, terminal is set to this value.

terminate (statement)

Use `terminate` to exit the Crosstalk application.

Format

```
terminate
```

`terminate` exits the Crosstalk application.

Note: To exit Crosstalk from the application, choose Exit from the File pull-down. ■

Crosstalk Mark 4 does not support this statement.

For related information, see the `quit` statement.

Example

```
clear
print "Crosstalk will terminate in 5 seconds"
for i = 1 to 5
   print at 5, 5, time(-1)
   wait 1 second
next
terminate
```

In this example, the script clears the window and then displays a message on the screen. Next, using the `for/next` construct, the script displays the current time once every second until 5 seconds have elapsed. Finally, it terminates Crosstalk.

time (function)

Use time to return a formatted time string.

Format

```
x$ = time(<integer>)
```

time returns the time in the correct format for the operating system country code.

integer is required; it is the number of seconds elapsed since midnight. You can use -1 as the argument to indicate the current number of elapsed seconds since midnight.

Note: If you want to check for a specific time, use the curhour, curminute, and cursecond functions. ■

Examples

```
print time(-1)
```

This example prints the current time.

```
x = time(32431)
```

In this example, the time represented by 32431 is returned in x.

```
open output "time.tst" as #1
write #1, "The file open time is " + time(-1)
while online
   string_in = nextline
   write line #1, string_in
wend
close #1
```

In this example, the file time.tst is opened for output, and a phrase is written to the file using the write statement. While the script is on line, each line of text from the host is written to the file. Then the file is closed.

timeout (system variable)

Use timeout to determine the status of the most recent wait or watch ... endwatch statement.

Format

```
timeout
```

timeout is true or false indicating whether the last nextline, wait, or watch ... endwatch statement timed out. timeout is true if the statement exceeded the time specification before finding the condition for which it was looking.

For related information, see the nextline, wait, and watch ... endwatch statements.

Example

```
repeat
   reply
   wait 1 second for "Login:"
until timeout = false
```

This example uses the timeout system variable and wait statement to log on to a host computer. The host, in this case, wants a number of carriage returns (CRs) so it can check the baud rate, parity, and stop bits. The CRs should be sent about once every second; and it will take an arbitrary number of CRs to wake up the host. When it is ready for your logon, the host sends the phrase "Login:"

trace (statement)

Use `trace` to trace how the lines in a script are executing.

Format

```
trace {on | off}
```

When `trace` is `on`, the script displays source script line numbers as the statements in the script are executed.

`trace` can be useful for debugging scripts.

For related information, see the `genlines` compiler directive.

Example

```
trace on
```

In this example, tracing is activated.

track (statement)

Use the track statement to watch for string patterns or keystrokes while on line.

Format

```
track <tracknum> <conditions>
```

The conditions are one or more of the following, separated by commas:

```
[[case] [space] <string>]
[quiet <time>]
[key <stroke_value>]
```

```
track routine <label or procedure>
```

```
track clear
```

The track statement lets you check for any number of events or incoming strings while the script is on line and then take some action based on which events occur. Use this statement with the wait and watch ... endwatch statements.

track events take precedence over wait and watch events. If a track event occurs while a script is at a wait or watch, the wait or watch is terminated and program control passes to the next statement. If you use track routine, control passes to the specified subroutine.

You can check events that you are tracking only at a wait or watch. If you do not use track routine, you will have to check the event with an if ... then ... else statement.

tracknum is the track number for the track statement. You can have any number of track statements active at one time. You can get an available track number with the freetrack function. Track numbers stay active as long as the script that set them is still running. When the script ends, the track numbers are closed.

When the string specified in *string* is received, the value of the corresponding track function is set to true.

There are a number of special sequences you can specify in *string*, each of which affects a track statement:

~_ (underscore)	Matches any white-space character.
~A	Matches any uppercase letter.
~a	Matches any lowercase letter.
~#	Matches any digit (0–9).
~X	Matches any letter or digit.
~?	Matches any single character.

A tilde (~) with a dash (-) followed by a special sequence character indicates that one or more occurrences of the sequence should be tracked. The following is an example:

~-#	Matches one or more occurrences of any digit (0–9)

time is a time expression in one of the following forms:

n hours
n minutes
n seconds
n ticks (1/10 seconds each)

Table 6-21 explains the track conditions.

Table 6-21. Conditions for the track statement

Condition	Explanation
string	case. Indicates that the string to be matched is case-sensitive. Unless this modifier is specified, Crosstalk ignores case. space. Indicates that *string* may contain white-space characters, such as spaces or tabs, that are significant. Crosstalk ignores white spaces unless this modifier is specified. **Note:** case and space can be used together to ensure an exact string match. ■

continued

Table 6-21. Conditions for the track statement (cont.)

Condition	Explanation
`quiet`	Indicates to wait until the communications line is quiet (no characters are received) for the amount of time specified in *time*.
`key`	Specifies a keyboard character to track. (See the `inkey` function earlier in this chapter for a list of keys and their corresponding numbers). Note that `key` comes from the local keyboard, not the communications line.

Use the `track routine` form of the `track` statement to designate a subroutine or a procedure that handles the `track` event.

Use the `track clear` form of the `track` statement to clear all tracked items and reset all of the track flags.

If you want to stop tracking a particular item, set the item to a null string. If you want to stop tracking everything, use `track clear`.

Note: You can use the `match` system variable to return the string found during the last `track` operation. ■

Versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information, see the `inkey`, `track`, and `freetrack` functions; the `match` system variable; and the `wait` and `watch ... endwatch` statements.

Example

```
track clear
track 1, space "system going down"
track 2, case space "no more messages"
track 3, case "thank YOU for calling"
track 4, key 833                -- Alt-A
track 5, quiet 1 minute
track routine check_track

wait for key 27                          -- Esc
...
...
end

label check_track
if track(1) then
   { bye : wait 8 minutes : call "megamail" : end }
if track(2) then goto send_outbound_messages
if track(3) then { bye : end }
if track(4) then end
if track(5) then { alarm 6 : reply : return }
```

This example shows `track` being used to watch for potential problems during an unattended, imaginary electronic mail session. `track` also looks for the ALT-A key identifier to indicate the script should end.

track (function)

Use the track function to determine if a string or event for which a track statement is watching has occurred.

Format

```
x = track
x = track(<tracknum>)
```

The track function checks if one of the strings or events for which a track statement is watching has been received and, if so, which one. Use this function with the wait and watch ... endwatch statements.

track events take precedence over wait and watch events. If a track event occurs while a script is at a wait or watch, the wait or watch is terminated and program control passes to the next statement. If you use track routine, control first passes to the specified subroutine.

You can check events that you are tracking only at a wait or watch. If you do not use track routine, you will have to check the event with an if ... then ... else statement.

tracknum is the track number for the track event.

The track function is set to true when the string or event in the corresponding track statement is received.

The first form of the track function returns the value of the lowest track number that has had an event occur. If none of the track statements has found a match, the track function returns false. The second form of the track function, track(*n*), returns true if the specified track event has occurred. Checking the function clears it.

Versions of Crosstalk for Windows older than 2.0 do not support this function.

For related information, see the freetrack function; the match system variable; and the track, wait, and watch ... endwatch statements.

Example

```
track 1, "System is going down"
wait for key 27
if track(1) then reply "logout"
```

In this example, the `track` statement is using track number 1 to watch for a string. The script is waiting for the ESC key. The `track` function for track 1 is checked to determine if the string was found, and if so, a logout message is sent to the host.

trap (compiler directive)

Use trap to control error trapping.

Format

```
trap {on | off}
```

trap enables and disables error trapping in a script. It allows you to control the actions of a script when errors are encountered that would normally stop script execution. When trap is on, it prevents an error condition from interrupting the running of a script.

The default setting for trap is off. When trap is on, the error function and the errclass and errno system variables should be tested to determine the occurrence, class, and number of an error. When the error function is tested for a value, it is cleared out. If it is not cleared, the next error that occurs will stop the script. Refer to error, errclass, and errno earlier in this chapter for more information on their use.

In general, it is best to set trap to on just prior to a statement that might generate an error and then set it to off immediately after the statement executes. Be sure to check the error return codes because a subsequent statement may reset the codes.

Example

```
string fname
fname = "*.exe"

trap on
send fname
trap off
if error then goto error_handler
```

In this example, the script branches to an error-handling routine if an error occurs when the send statement is executed.

true (constant)

Use `true` to set a variable to logical true.

Format

```
x = true
```

`true` is always logical true. `true`, like its complement `false`, exists as a way to set variables on and off. If `true` is converted to an integer, its value is 1 (one).

For related information, see the `false`, `on`, and `off` constants.

Example

```
x = 1
done = false
while not done
   x = x + 1
   if x = 10 then done = true
wend
```

In this example, the statements in the `while/wend` construct are repeated until `done` is `true`.

unloadallquickpads (statement)

Use `unloadallquickpads` to unload all of the QuickPads for the current session.

Format

```
unloadallquickpads
```

This statement unloads all open QuickPads for the current session.

Note: The QuickPads for the session must already be loaded using the `loadquickpad` or `loadallquickpads` statement. ■

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information, see the statements `loadallquickpads`, `loadquickpad`, and `unloadquickpad`.

Example

```
unloadallquickpads
```

unloadquickpad (statement)

Use `unloadquickpad` to unload the specified QuickPad for the current session.

Format

```
unloadquickpad <string>
```

This statement unloads the QuickPad specified in *string*.

Note: The QuickPad for the session must already be loaded using the `loadquickpad` or `loadallquickpads` statement. ■

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information, see the statements `loadallquickpads`, `loadquickpad`, and `unloadallquickpads`.

Example

```
unloadquickpad "apad"
```

In this example, the QuickPad "apad" is unloaded.

upcase (function)

Use upcase to convert a string to uppercase letters.

Format

```
x$ = upcase(<string>)
```

upcase converts only the letters a–z to uppercase characters; numerals, punctuation marks, and notational symbols are unaffected.

For related information, see the lowcase function.

Example

```
string yn
print "Do this again?";
input yn
if upcase(yn) = "Y" then goto start
```

In this example, the character entered by the user, which is stored in the yn variable, is checked to determine if it is an uppercase "Y." If it is, the script branches to the label start.

upload (statement)

Use `upload` to upload a text file.

Format

`upload <filename>`

filename is the name of an existing ASCII text file.

Use this command only when you are on line to the host.

Note: To initiate a file upload using your Crosstalk application, choose Session from the Action pull-down and then choose Upload Text File. ■

Refer to your Crosstalk user's guide for more information about uploading ASCII text files.

Examples

```
upload "login.xws"
```

In this example, the script uploads a file called `login.xws`.

```
upload fname
```

In this example, the script uploads the file assigned to the `fname` variable.

userid (system variable)

Use `userid` to read or set a user account number or identifier for a session.

Format

```
userid = <string>
```

`userid` sets and reads the user account identification associated with the current session. `userid` is limited to 40 characters.

Note: To set up the equivalent parameter using your Crosstalk application, choose Session from the Settings pull-down. Then choose the General icon and modify the User ID parameter. ■

Examples

```
userid = "76004,302"
```

In this example, `userid` is set to the specified string.

```
reply userid
```

In this example, `userid` is sent to the host.

```
userid = ""
```

In this example, `userid` is cleared.

val (function)

Use `val` to return the numeric value of a string.

Format

```
x = val(<string>)
```

The `val` function, like the `intval` function, returns a numeric value; however, `val` returns a real (floating point) number rather than an integer. The `val` function evaluates *`string`* for its numerical meaning and returns that meaning as a real. Leading white-space characters are ignored, and *`string`* is evaluated until a non-numeric character is encountered.

The characters that have meaning to the `val` function are: "0" through "9", ".", "e", "E", "-", and "+".

Versions of Crosstalk for Windows older than 2.0 do not support this function.

For related information, see the `intval` function.

Example

```
num = val(user_input_string)
```

In this example, `user_input_string` is converted to a real number and returned in `num`.

version (function)

Use `version` to return the Crosstalk version number.

Format

```
x$ = version
```

`version` returns the Crosstalk version number as a string.

Example

```
print version
```

In this example, the Crosstalk version number is displayed.

wait (statement)

Use `wait` to wait for a string of text from the communications device or to wait for a keystroke.

Format

```
wait [<time>] for <conditions>
```

The conditions are one or more of the following, separated by commas:

```
[[case] [space] <string>]
[quiet <time>]
[key <key_value>]
[count <integer>]
```

The `wait` statement waits the amount of time specified in *time* for one of the values specified in the foregoing format.

There are a number of special sequences you can specify in *string*, each of which affects a `wait` statement. See the `track` statement earlier in this chapter for a list of applicable sequences.

time is a time expression in one of the following forms:

n hours
n minutes
n seconds
n ticks (1/10 seconds each)

Table 6-22 explains the `wait` conditions.

Table 6-22. Conditions for the wait statement

Condition	Explanation
string	`case`. Indicates that the string to be matched is case-sensitive. Unless this modifier is specified, Crosstalk ignores case. `space`. Indicates that Crosstalk should match all white-space characters exactly as specified in *string*. Any extra white-space characters are not allowed. **Note:** `case` and `space` can be used together to ensure an exact string match. ■
`quiet`	Indicates to wait until the communications line is quiet (no characters are received) for the amount of time specified in *time*.
`key`	Specifies a keyboard character for which to wait. (See the `inkey` function earlier in this chapter for a list of keys and their corresponding numbers). `key 0` means wait for any key. You can retrieve the value of the key that was pressed by using the `match` function. Note that `key` comes from the local keyboard, not from the communications line.
`count`	Indicates to wait for the number of characters specified in *integer*.

If one of the *time* options (`minutes`, `seconds`, or `ticks`) is specified, and the specified string is not matched, the `timeout` system variable returns `true`, indicating that the desired string was not received in the time specified. The default time is forever.

Only the following constructs are valid when the session is off line; the session must be on line to use any other option.

```
wait <time>
wait for key <inkey_value>
wait <time> for key <inkey_value>
```

For related information, see the `match` and `timeout` system variables, the `track` and `watch` ... `endwatch` statements, and the `inkey` function.

Examples

```
wait for "Login:" : reply userid
```

In this example, the script waits for the specified phrase and sends the information stored in the `userid` system variable to the host.

```
wait 1 second for "Hello"
```

In this example, the script waits 1 second for the specified phrase.

```
wait for "A", "B", "C"
string_in = match
case string_in of
   "A" : reply 'We received an "A"'
   "B" : reply 'We received a "B"'
   "C" : reply 'We received a "C"'
endcase
```

In this example, the scripts waits for any one of the characters "A," "B," or "C." Depending on which value is received, the appropriate response is sent to the host.

```
wait 20 seconds for "in:" : if timeout then
   goto no_ans
```

In this example, the script waits 20 seconds for a phrase. If the phrase does not arrive within the 20-second time frame, the script branches to the label `no_ans`.

```
wait for count 10
```

In this example, the script waits until 10 characters are received.

```
wait for case "UserID:"
```

In this example, the script must wait for an exact upper- and lowercase match for the `UserID:` prompt.

watch ... endwatch (statements)

Use watch ... endwatch to watch for one of several strings of text from the communications device or to watch for a keystroke.

Format

```
watch [<time>] for
   [[case] [space] <string> :
      [<statement group>]]
   [quiet <time>] : [<statement group>]
   [key <stroke_value>] : [<statement group>]
   [count <integer>] : [<statement group>]
endwatch
```

The watch statement waits the length of time specified in *time* for one of the conditions specified in the foregoing format. *time* is optional; however, if you do not specify a time limit, watch ... endwatch waits forever.

watch performs the statements in *statement group* when the corresponding condition is met. The program logic then continues with the statement following endwatch. *statement group* is optional.

string, quiet, and so on, are conditions for which to watch.

There are a number of special sequences you can specify in *string*, each of which affects a watch statement. See the track statement earlier in this chapter for a list of applicable sequences.

time is a time expression in one of the following forms:

n hours
n minutes
n seconds
n ticks (1/10 seconds each)

Table 6-23 explains the watch conditions.

Table 6-23. Conditions for the watch statement

Condition	Explanation
string	case. Indicates the case of the string must be matched exactly. watch is case-insensitive unless the case keyword is used. space. Indicates the string cannot contain extra white-space characters. watch is not sensitive to embedded white-space characters unless the space keyword is used. **Note:** case and space can be used together to ensure an exact match. ■
quiet	Indicates the communications line must remain quiet (no characters should be received) for the amount of time specified.
key	Specifies a keyboard key for which to watch. (See the inkey function earlier in this chapter for a list of keys and their corresponding numbers.)
count	Specifies to watch for the number of characters given in *integer*.

The watch/endwatch construct is not a looping construct. When one of the watch conditions is met, the script goes on to execute the appropriate statement(s). If you want to use these statements in a loop, place them inside a while/wend construct.

Use this statement only when you are on line, unless you are using it to watch for a keystroke.

For related information, see the track, wait, and while ... wend statements; the match system variable; and the inkey function.

Examples

```
watch for
    "Login:" : goto login_procedure
    "system down" : goto cant_log_in
    quiet 10 minutes : goto system_is_dead
    key 27 : reply "logoff" : bye : end
endwatch
```

In this example, the script watches for one of the specified events. If any one of the events is true, the statement(s) to the right of the colon are executed, and the `watch/endwatch` construct is completed.

```
while online
    watch for
        "graphics" : reply "Yes"
        "first name" : reply userid
        "password" : reply password : end
    endwatch
wend
```

This example shows how to make the `watch/endwatch` construct part of a `while/wend` loop. The code shown is a simple login script for the Crosstalk BBS. The `while/wend` construct continues to loop until `watch` receives the `password:` prompt.

weekday (function)

Use `weekday` to return the number of the day of the week.

Format

```
x = weekday[(<integer>)]
```

`weekday` returns the number (0–6) of the current day of the week. Sunday is day 0 (zero), Monday is 1, and so on.

If *integer* is specified, `weekday` returns the day of the week for a given date in the past or future.

Examples

```
print weekday, weekday(365)
```

For a Friday, the script in this example prints 5, a tab, and 1.

```
print weekday(filedate("somefile"))
```

This example shows how to print the number of the day of the week when `somefile` was last modified.

while ... wend (statements)

Use `while ... wend` to perform a statement or group of statements as long as a specified condition is true.

Format

```
while <expression>
    ...
    ...
    ...
wend
```

expression is any logical expression; it can be a combination of numerical, boolean, or string comparisons that can be evaluated as either `true` or `false`.

`while` lets you perform one or more statements as long as a certain expression is `true`. Unlike the `repeat/until` construct, the `while/wend` construct is not executed at all if the expression is `false` the first time it is evaluated.

`wend` indicates the end of the conditional statements.

When using any looping construct, be sure the terminating condition (that is, *expression*) will eventually become `true`, or that there is some other exit from the loop.

For related information, see the `repeat ... until` statements.

Example

```
x = 1
while x <> 100
    print x
    x = x + 1
wend
```

In this example, the script prints the numbers 1 through 99.

winchar (function)

Use winchar to return the ASCII value of a character read from a session window.

Format

```
x = winchar(<row, col>)
```

winchar reads a character from a window, at *row*, *col*. The winchar function helps you determine the results of operations not under script control, such as the appearance of a certain character at a certain location on the screen while under the control of a host computer.

For related information, see the nextchar, nextline, and winstring functions.

Example

```
char1 = winchar(1, 1)
```

In this example, the character at row 1, column 1 is stored in char1.

winsizex (function)

Use `winsizex` to return the number of visible columns in the session window.

Format

```
x = winsizex
```

`winsizex` returns the width of the session window, in columns. This function is especially handy when writing scripts that display information and need to accommodate the size of the terminal screen.

For related information, see the `winsizey` function.

Examples

```
print winsizex
```

In this example, the script prints the width, in columns, of the terminal window at its current size.

```
if winsizex < 80 then maximize
```

If the session window is less than 80 columns in width, this statement maximizes it.

winsizey (function)

Use `winsizey` to return the number of visible rows in the session window.

Format

```
x = winsizey
```

`winsizey` returns the height of the session window, in rows. This function is especially useful in scripts that must accommodate the screen size to operate properly.

For related information, see the `winsizex` function.

Example

```
if winsizey < 24 then maximize
```

If the session window is less than 24 rows in length, this statement maximizes it.

winstring (function)

Use `winstring` to return a string read from a session window.

Format

```
x$ = winstring(<row, col, len>)
```

`winstring` reads a string of characters from the session window, beginning at *row*, *col*, for *len* characters, with any trailing spaces removed.

`winstring` lets you determine the results of operations not under script control, such as the appearance of a certain string at a certain location on the screen while under the control of a host computer.

Example

```
string data
data = winstring(10, 10, 11)
if data = "Login name:" then reply userid
```

In this example, the script's `data` variable is assigned the contents of the screen area specified by the `winstring` function. If those characters equal "Login name:" then the `userid` system variable is sent to the host.

Win winversion (function)

Use `winversion` to check the Windows version number.

Format

```
x$ = winversion
```

`winversion` returns the Windows version number as a string.

Example

```
print winversion
```

In this example, the script displays the Windows version number on the screen. ■

write (statement)

Use write to write data to a sequential disk file.

Format

```
write [#<filenum>, ] [<item>] [{, | ;} ...
   [<item>]] ... [;]
```

The write statement operates only on files opened in output or append modes. *filenum* must be an open file output number; if *filenum* is not specified, the default output file number, which is stored in the variable defoutput, is assumed.

The write statement writes lines containing comma-delimited fields of ASCII data. Each write adds the members of *string_var_list* to the file, with the contents of each member separated from the next by a comma. To suppress the commas in the output file, separate the items in the list with semicolons instead of commas. If the contents of a member of *string_var_list* include commas or quotation marks, use the quote function to enclose the members in appropriate quotation marks.

Normally, write terminates each write to the file with a carriage-return/line-feed (CR/LF) pair. To suppress the CR/LF, use the trailing semicolon.

For related information, see the defoutput system variable, the open and write line statements, and the quote function.

Examples

```
open output file_name as #1
write #1, alpha, beta, gamma;
close #1
```

In this example, the script opens a file, writes the specified strings of data to the file, and closes the file.

```
write #1, quote(var1), quote(var2), ...
   quote(var3)
```

In this example, the script encloses the data strings in quotation marks before writing them to the file.

write line (statement)

Use write line to write data to a sequential disk file.

Format

```
write line [#<filenum>, ] [<item>] [{, | ;} ...
   [<item>]] ... [;]
```

As with the write statement, the write line statement operates only on files opened in output or append modes. *filenum* must be an open file output number; if *filenum* is not specified, the default output file number, which is stored in the defoutput system variable, is assumed.

The write line statement writes a new line for each item. You can suppress this by separating items with a semicolon.

Normally, write line terminates each write to the file with a carriage-return/line-feed (CR/LF) pair. To suppress the CR/LF, use the trailing semicolon.

For related information, see the defoutput system variable and the open and write statements.

Examples

```
write line "end of test"
```

In this example, the text line "end of test" is written to a file. Since the file number is not specified, the default file number in defoutput is used.

```
write line #1, some_text
```

In this example, the script writes the contents of some_text to the file identified by the file number #1.

xpos (function)

Use xpos to find out the column location of the cursor.

Format

```
x = xpos
```

xpos returns the number of the column on which the cursor rests.

Examples

```
cur_col = xpos
```

In this example, the script assigns the cursor's current column position to the cur_col variable.

```
if xpos = winsizex - 1 then alarm
```

In this example, the terminal sounds an alarm if the cursor position is one column less than the size of the window.

ypos (function)

Use `ypos` to find out the row location of the cursor.

Format

```
x = ypos
```

`ypos` returns the number of the row on which the cursor rests.

Examples

```
cur_row = ypos
```

In this example, the script assigns the cursor's current row position to the `cur_row` variable.

```
if ypos = winsizey - 1 then alarm
```

In this example, the terminal sounds an alarm if the cursor position is one row less than the size of the window.

zoom (statement)

Use zoom to enlarge a session window to the size of the Crosstalk application window.

Format

```
zoom
```

zoom enlarges a session window to fill the Crosstalk application frame.

Crosstalk Mark 4 and versions of Crosstalk for Windows older than 2.0 do not support this statement.

For related information, see the hide and show statements.

Example

```
if online then
   zoom
```

In this example, the session window is enlarged if the session is on line to the host.

7 Working with Terminal, Connection, and File Transfer Tools

The tool concept 7-2

Terminal tool 7-3

Connection tool 7-4

File transfer tool 7-5

The tool concept

A tool is a Crosstalk code file that is used to control a specific aspect of a communications session. There are three types of tools: terminal, connection, and file transfer. Each tool type offers a number of individual tools, and each of those tools is suited to a specific communications task. Only one tool of each type is used for any given session.

You do not need to use each type of tool to complete a communications task. At a minimum, communications requires a connection tool and a terminal tool; a file transfer tool is needed only when you want to transfer files. For example, if you are simply calling an information service to browse the news, all you need is a connection tool appropriate for your communications hardware and a terminal tool appropriate for the system with which you are communicating.

You can establish the settings for the various tools using the Connection, Terminal, and File Transfer Tools provided with your software. You can also set up or modify these settings in your scripts. The following sections provide information you need in order to work with the three types of tools.

Terminal tool

The remote systems with which you communicate are designed to be connected to terminals of their own system type. This means they expect to interact with specific terminals whose keyboard and display characteristics are not exactly the same as that of a PC. During communications with a remote system, the terminal tool causes your PC to emulate (assume the characteristics of) a terminal of the correct type. This allows communications to continue just as if you were using a terminal designed specifically for that remote system.

The terminal tool options provided with the software are set to the defaults of an actual terminal. Even though many options are available to ensure complete emulation capabilities, you do not need to be concerned with all of the possible settings because the default settings allow communications to continue normally with most remote systems. In general, you would change the default values only if the remote system has been configured to require specific settings for its terminals or if an option suits your personal preference.

Two fonts are included with your product: the IBM-PC font and the DCA DEC font. These fonts are in two forms—bitmap and True Type. Crosstalk automatically selects the correct font for the terminal tool you are using. For example, the DCA DEC font is used for DEC, HP, and WYSE emulations, and the IBM-PC font is used for IBM-PC (ANSI) emulation. IBM 3101, TTY, Vidtex, and FTTERM emulations can use any of the fonts provided, including the DEC and IBM-PC fonts. You can override the default font, but incorrect characters may result.

Mac

Crosstalk for Macintosh, because of its support for the Apple Comm ToolBox, can use third-party terminal tools that are not shipped with your Crosstalk product. ■

To set up or modify the terminal emulation type in a script, you must use the `assume` statement to access the terminal tool variables and then assign the appropriate terminal emulation name to the `terminal` system variable. For information about the `assume` statement and the `terminal` system variable, refer to Chapter 6, "Using the Programming Language."

Note: To find detailed information about the terminal tool variables, refer to the on-line help available for the Terminal tool. ■

Connection tool

The connection tool contains the settings that control the hardware device used for communications. These settings determine such characteristics as communications speed, the character format of transmitted data, and flow control.

Mac

Crosstalk for Macintosh provides the Apple Serial and Apple Modem tools with the software. You can also use tools from other vendors, including Apple's LAT tool, the Hayes modem tool, and other tools that support the CTB standard. ■

Win

Crosstalk for Windows provides tools that support direct connection with no modem (Local COM Port), dialing a modem attached to your PC (Local Modem), dialing a modem attached to a NetWare Asynchronous Communications Server (NASI-Advanced and NASI-Basic), and INT 14. ■

To set up or modify the connection device type in a script, you must use the `assume` statement to access the connection tool variables and then assign the appropriate connection device name to the `device` system variable. For information about the `assume` statement and the `device` system variable, refer to Chapter 6, "Using the Programming Language."

Note: To find detailed information about the connection tool variables, refer to the on-line help available for the Connection tool. ■

File transfer tool

The file transfer tool specifies a file transfer protocol, which is a standardized method of exchanging files between two computers. Each file transfer protocol has a unique set of rules and conventions that define, among other things, the number of bytes to send for each block of data and how to detect and correct errors.

For a file transfer to work, both the sending and receiving computer must use the same protocol. To ensure maximum flexibility with a variety of remote systems, Crosstalk supports the most common file transfer protocols.

Mac

Crosstalk for Macintosh, because of its support for the Apple Comm ToolBox, can use third-party file transfer protocol tools that are not shipped with your Crosstalk product. ■

To set up or modify the file transfer protocol in a script, you must use the `assume` statement to access the file transfer tool variables and then assign the appropriate file transfer protocol name to the `protocol` system variable. For information about the `assume` statement and the `protocol` system variable, refer to Chapter 6, "Using the Programming Language."

Note: To find detailed information about the file transfer tool variables, refer to the on-line help available for the File Transfer tool. ■

8 Compatibility Issues

Introduction	8-2
Crosstalk for Windows	8-2
Crosstalk for Macintosh	8-4
Crosstalk Mark 4	8-4

Introduction

The language elements presented in this guide are applicable to scripts developed for Crosstalk for Windows or Crosstalk for Macintosh. However, many of the elements are also valid for Crosstalk Mark 4. This chapter explains the CASL compatibility among these Crosstalk applications.

Crosstalk for Windows

There are differences between this implementation of CASL and that used in older versions of Crosstalk for Windows. The following sections list the language elements that have been added to, changed for, and removed from this release of CASL.

New elements

The following new language elements are supported only for Crosstalk for Windows, version 2.0 and newer:

activatesession
activesession
assume
case/endcase
connectreliable
copy
ddeack
ddeadvise
ddeadvisedatahandler
ddenak
ddeunadvise
device
dialmodifier (Was modifier)
do
downloaddir
exit (from a procedure)
for/next
func/endfunc
genlabels
genlines
hideallquickpads
hidequickpad
keys
loadquickpad
max (Was an operator)
min (Was an operator)
nextline function
on
perform
press
proc/endproc
return (from a function)
rewind
scriptdesc
session
sessname
sessno
showallquickpads
showquickpad
stroke
track function
track statement
unloadallquickpads
unloadquickpad
val
zoom

Changed elements

The following language elements have changed for Crosstalk for Windows, version 2.0 and newer:

```
backups
chain
close
cmode
connected
dialogbox/enddialog
display
fileattr
filedate
filesize
filetime
get
go
kermit
netid
number
open
pack
password
pad
printer
protocol
quit
read
read line
redialcount
redialwait
script
startup
write
write line
time
terminal
tabex
userid
```

Removed elements

The following language elements are no longer supported for Crosstalk for Windows, version 2.0 and newer:

```
answersetup
bookname
colorscreen
connectarq
connectspeed
dial
dialprefix
dialsuffix
dirxwp
dirxws
fkey statement
fkey function
hostmode
hostscript
inbook
kclear
ldnumber
misc
outnumber
review
secret
showactive
showhscroll
showinput
showactive
showkeybar
showstatusbar
showvscroll
windowwrap
```

Crosstalk for Macintosh

The following language elements are not supported for Crosstalk for Macintosh:

```
chmod
curdrive
ddeack
ddeadvise
ddeadvisedatahandler
ddeexecute
ddeinitiate
ddenak
ddepoke
dderequest
ddestatus
ddeterminate
ddeunadvise
dosversion
drive
environ
fncheck
fnstrip
move
size
winversion
```

Crosstalk Mark 4

The following language elements are not supported for Crosstalk Mark 4:

```
activate
activatesession
activesession
alert
connectreliable
ddeack
ddeadvise
ddeadvisedatahandler
ddeexecute
ddeinitiate
ddenak
ddepoke
dderequest
ddestatus
ddeterminate
ddeunadvise
dialogbox/enddialog
dirfil
func/endfunc
footer
header
hideallquickpads
hidequickpad
kermit
loadquickpad
max
maximize
message
min
minimize
on
restore
return (from a function)
showallquickpads
showquickpad
tabwidth
terminate
unloadallquickpads
unloadquickpad
winversion
zoom
```

A Windows Considerations

Developing DDE Scripts A-2

CASL DDE commands A-8

Developing DDE scripts

Scripts developed for Crosstalk for Windows can exchange information with other applications using a protocol called Dynamic Data Exchange. Using DDE, you can transfer data on a one-time basis, or establish an ongoing dialog with other applications. This section explains things to keep in mind when using DDE to communicate with other applications.

Topic name support

To execute a Crosstalk command from another application during a DDE conversation, use "XTALK" as the application name, and "system" as the topic. If a topic name is not specified, it is treated as "system."

Crosstalk also accepts a session name as a DDE topic. With the additional session topic, you can access Crosstalk by referencing the name of a session. The session name is displayed in the session window title bar.

Requesting information

The remote application can execute several requests during a DDE conversation. Table A-1 lists valid requests for the system topic.

Table A-1. Valid requests for the system topic

Request	Crosstalk response
topics	Returns a space-separated list of open profile items.
status	Returns the word "Ready" or "Busy," depending on the application status.
formats	Returns the numeric value of the Windows define CF_TEXT.
sysitems	Returns a list of the requests described in this table.

Table A-2 lists valid requests for a session topic.

Table A-2. Valid requests for a session topic

Request	Crosstalk response	
`status`	Returns one of the following:	
	Busy	Connecting or disconnecting.
	Disconnected	Not connected.
	Ready	Connected but not busy.
	Script	A script is running. (A `ddeexecute` command will fail.)
	Transfer	A file transfer is in progress. (A `ddeexecute` command will fail.)
A `public` variable	Returns the requested variable.	

Executing Crosstalk commands

There are several Crosstalk commands you can execute from other applications during a DDE conversation. You should enclose the commands in brackets. For example, the following command instructs Crosstalk to dial the CSERVE session:

```
"[dial(CSERVE)]"
```

Table A-3 lists valid commands for the `system` topic.

Table A-3. Valid commands for the system topic

Command	Description
`[load(<entry_name>)]`	Starts the specified session. A topic by this name is created.
`[new]`	Creates a new untitled session. The session topic name may vary depending on how many untitled sessions are already open.

Table A-4 lists valid commands for a session topic.

Table A-4. Valid commands for a session topic

Command	Description
`[bye]`	Disconnects the connection. This command is equivalent to the CASL `bye` statement.
`[cancel]`	Cancels the currently running script. This command is equivalent to the CASL `halt` statement.
`[close]`	Requests Crosstalk to terminate. Termination is delayed until the DDE channel is closed. Be careful in using this command; when Crosstalk receives a `close` command, it terminates even if a connection is active.
`[dial(<entry_name>)]`	Loads and dials the specified session. The script associated with the session (if any) is run after a connection is made. This command is NOT valid if a CASL script is running. The command is equivalent to the `[load (<entry_name>)] [go]` command combination.

continued

Table A-4. Valid commands for a session topic (cont.)

Command	Description
[execute(<*script_name*>)]	Executes the specified script. The script's name can include arguments for the script. This command is NOT valid if a CASL script is running.
[go]	Connects to the selected communications port. This command is equivalent to the CASL `go` statement.
[load(<*entry_name*>)]	Starts the specified session. This command is NOT valid when a CASL script is running.
[new]	Loads default Crosstalk parameters, and starts the NORMAL session. You can use this command to reset Crosstalk settings. This command is NOT valid when a CASL script is running.
[save]	Saves Crosstalk settings using the current session.
[saveas(<*entry_name*>)]	Saves the Crosstalk settings using the session name specified in the command.

Learning more about DDE

Several DDE scripts are provided with the Crosstalk for Windows software. If you are not familiar with DDE, you can run these scripts to learn about it. If you have Microsoft Excel, you can use the Excel demonstration scripts, also provided with the software, to see how Crosstalk and Excel interact through DDE.

DDE demonstration scripts

The DDE demonstration scripts place stock price information in an Excel spreadsheet. Two demonstration scripts are available: an on-line script and an off-line script. The on-line script accesses CompuServe's stock price information to place current stock prices in a chart. To run the on-line script, you must have a CompuServe account and be able to edit session information. The off-line script simulates this process and does not require a CompuServe account.

Table A-5 lists the files that make up the DDE demonstration script set.

Table A-5. DDE demonstration script files

File name	Purpose
EXCELSTK.XWS	This is the on-line script. It runs after a connection with CompuServe is established. Its purpose is to extract current stock data, which is passed to Excel through DDE. This script uses the CSERVE.XWP session.
EXCELOFF.XWS	This is the off-line script. It sends simulated stock data to Excel through DDE.
XTALKDDE.XLM	This is the Excel macrosheet. It opens automatically.
XTALKDDE.XLS	This is the Excel worksheet. It opens automatically.
XTALKDDE.XLW	This is the Excel workspace. It is the file you open from Excel.

Running the DDE scripts

You must start both the on-line and off-line scripts from Excel. To do this, follow these steps:

1. Start Excel and maximize the window for best display.
2. Choose Open from the File pull-down.
3. Specify XTALKDDE.XLW as the file to open.

Use the keys shown in Table A-6 to run the on-line or off-line script or to display help information.

Table A-6. DDE demonstration script control keys

Keys	Action
CTRL-A	Runs the on-line script.
CTRL-Z	Runs the off-line script.
CTRL-H	Displays help information.

Information provided for DDE commands

Before you refer to the DDE commands in the sections that follow, you may find it helpful to understand how the information is presented. The command names are presented in alphabetical order. For each command, the format of the command is shown, followed by an example of how you can use the command in your script.

ddeack (statement)

Use ddeack to send a positive acknowledgment to the application that sent a ddeadvisedata message.

Format

ddeack <*ddechannel*>

ddechannel is the integer DDE channel number. This variable should be defined at the beginning of the script. Windows assigns a value to the variable when you initiate a DDE conversation. See the ddeinitiate statement later in this chapter for more information.

You must use this command inside your ddeadvisedata event handler; otherwise, a run-time error occurs.

Versions of Crosstalk for Windows older than 2.0 do not support this statement.

Example

```
ddeack dde_channel
```

In this example, an acknowledgment of receipt of a ddeadvisedata message is sent through the channel dde_channel.

ddeadvise (statement)

Use ddeadvise to request notification of all changes to a specified data item. The request remains in effect until it is canceled with the ddeunadvise statement.

Format

```
ddeadvise <ddechannel>, <itemname>
```

ddechannel is the integer DDE channel number. This variable should be defined at the beginning of the script. Windows assigns a value to the variable when you initiate a DDE conversation. See the ddeinitiate statement later in this chapter for more information.

itemname is the name of the data item about which you want to be informed.

Versions of Crosstalk for Windows older than 2.0 do not support this statement.

Example

```
ddeadvisedatahandler ddeadvisedataprocedure
ddeadvise excelID, "R4C5"
```

In this example, the DDE data handler ddeadviseprocedure is enabled. Then a ddeadvise request is sent for the item R4C5.

ddeadvisedatahandler (event handler)

Use ddeadvisedatahandler to enable the event handler that will handle ddeadvisedata message events This type of event occurs when an incoming ddeadvisedata message is received.

Format

```
ddeadvisedatahandler [<ddeadvisedatahandlername>]
```

You must declare your event handler before you enable it. Declare the ddeadvisedatahandler procedure as follows:

```
proc <ddeadvisedatahandlername> ...
   integer <ddechannel>, string <itemname>, ...
   string <data>
   ...
   ...
endproc
```

This procedure must accept three arguments: *ddechannel* (the channel through which the advise notification is received), *itemname* (the name of the data item about which you asked to be informed), and *data* (the data in *itemname* that has changed). No additional ddeadvisedata messages are processed until this procedure returns control.

At some point in your event handler, you should reply using either ddeack for a positive acknowledgment or ddenak for a negative acknowledgment.

Note: If you want to turn off ddeadvisedata message handling, use ddeadvisedatahandler without specifying a procedure name. When you omit the procedure name, the CASL default DDE advise handler, which ignores ddeadvisedata events, becomes active. ■

Versions of Crosstalk for Windows older than 2.0 do not support this event handler.

Example

```
proc ddeadvisedataprocedure integer dde_channel, ...
    string itemname, string data
    ...
    ...
endproc
...
...
ddeadvisedatahandler ddeadvisedataprocedure
```

In this example, the advise handler `ddeadvisedataprocedure` is declared, and then it is enabled.

ddeexecute (statement)

Use ddeexecute to ask another application to execute a command.

Format

```
ddeexecute <ddechannel>, <command>
```

ddechannel is the integer DDE channel number. This variable should be defined at the beginning of the script. Windows assigns a value to the variable when you initiate a DDE conversation. See the next statement, ddeinitiate, for more information.

command must be a string expression. The DDE protocol recommends that all applications use the following format for commands:

```
<commands>    = [<command>]...
<command>     = <operation>[(<arguments>)]
<arguments>   = <argument>[, <argument>]...
```

Example

```
[open("sales.xls")] [print]
```

In this example, there are two commands: the first command consists of the operation open, with its string argument sales.xls; and the second command is the operation print. Note that commands are enclosed in square brackets; and argument(s), which are optional, are enclosed in parentheses.

Suppose you have initiated a DDE conversation to Excel, and you want to send the message in the preceding example. Write the command as follows:

```
ddeexecute excelid, '[open("sales.xls")]' + ...
   '[print]'
```

ddeinitiate (statement)

Use ddeinitiate to open a DDE conversation with another application. If more than one application responds to the ddeinitiate request, the conversation is set up with the first response received.

Format

```
ddeinitiate <ddechannel>, <applicationname>, ...
   <topicname>
```

The ddeinitiate statement opens a DDE conversation with a specified application. If ddeinitiate fails to establish the conversation because the other application is not running, a run-time error occurs. You can use the trap compiler directive to trap the error and then use the run statement to start the application. For more information about trap and run, see Chapter 6, "Using the Programming Language."

ddechannel is the DDE channel used to communicate with the other application. Windows assigns a value to this variable when you initiate a DDE conversation. You must declare the variable as an integer before you use the ddeinitiate statement. Other DDE statements covered in this chapter also use the *ddechannel* variable.

Note: You can open DDE channels to more than one application, provided that each ddeinitiate statement uses a unique variable name for *ddechannel*. ■

The application is identified by *applicationname*. This is the application's DDE name. Refer to your DDE documentation for appropriate names.

The topic is identified by *topicname*. The value used for this variable is only meaningful to the other application. Refer to the application DDE documentation to find valid topic names.

Example

```
integer dde_channel
ddeinitiate dde_channel, "Excel", "System"
```

In this example, the variable `dde_channel` is declared as an integer. The variable is then used in the `ddeinitiate` statement to establish a conversation with the application `"Excel"` and the topic `"System."`

ddenak (statement)

Use ddenak to send a negative acknowledgment to the application that sent a ddeadvisedata message.

Format

```
ddenak <ddechannel>
```

ddechannel is the integer DDE channel number. This variable should be defined at the beginning of the script. See ddeinitiate earlier in this chapter for more information.

You must use this command inside your ddeadvisedata event handler; otherwise, a run-time error occurs.

Versions of Crosstalk for Windows older than 2.0 do not support this statement.

Example

```
ddenak dde_channel
```

In this example, a negative acknowledgment, indicating that a ddeadvisedata message was not accepted is sent through the channel dde_channel.

ddepoke (statement)

Use ddepoke to send a string of data to the application at the other end of a DDE conversation.

Format

```
ddepoke <ddechannel>, <itemname>, <data>
```

This statement sends a message by way of *ddechannel* to the other application, requesting the application to assign the value in *data* to *itemname*.

ddechannel is the channel used to communicate with the application. You should define this variable at the beginning of your script. For more information, see ddeinitiate earlier in this chapter.

itemname is the name of the variable in the remote application that is to contain the data string. If you do not know the name of the variable, check the documentation for the remote application.

data is the data string the other application should assign to *itemname*.

Example

```
ddepoke dde_channel, "user_name", "chuck"
```

In this example, the script sends the string "chuck" to the other application, using the channel dde_channel. The other application assigns "chuck" to user_name.

dderequest (statement)

Use dderequest to request data from another application.

Format

dderequest <*ddechannel*>, <*remoteitem*>, <*myitem*>

This statement sends a request through the *ddechannel* asking the other application to return the value of *remoteitem* in *myitem*.

ddechannel is the DDE channel used to communicate with the other application. You should define this variable at the beginning of your script. See ddeinitiate earlier in this chapter for more information.

remoteitem is the name of the other application's variable; it contains the value to be returned to *myitem*.

myitem is the name of the string variable in your script that is to contain the data received from the other application.

Example

```
string cellA1
dderequest dde_channel, "R1C1", cellA1
```

In this example, the variable cellA1 is declared as a string. Then the script sends a dderequest asking the other application to send the data in "R1C1" to the script's variable cellA1.

ddestatus (function)

Use ddestatus to check whether a DDE channel is open.

Format

```
x = ddestatus(<ddechannel>)
```

The ddestatus function returns a true or false value indicating whether the DDE channel is open. Use this function to periodically check the status of a previously opened DDE conversation.

ddechannel is the DDE channel used to communicate with the other application. Windows assigns a value to the variable when you initiate a conversation with another application. For more information, see the ddeinitiate statement earlier in this chapter.

Example

```
boolean x
x = ddestatus(dde_channel)
print "DDE Status = "; x
```

In this example, x is declared as a boolean variable. The ddestatus function returns a true or false value in x. The print statement then prints the value in x.

ddeterminate (statement)

Use `ddeterminate` to close a DDE conversation.

Format

```
ddeterminate <ddechannel>
```

ddechannel is the DDE channel used to communicate with the other application. Its value is set by Windows when you initiate a DDE conversation. For more information, see the `ddeinitiate` statement earlier in this chapter.

Example

```
ddeterminate dde_channel
```

In this example, you close the channel `dde_channel`.

ddeunadvise (statement)

Use ddeunadvise to cancel a request made previously with the ddeadvise procedure. When you use this procedure, you send a request asking to no longer be informed of changes either to a particular data item or to any data item for which ddeadvise requests have been made.

Format

```
ddeunadvise <ddechannel>, <itemname>
ddeunadvise <ddechannel>
```

Use the first form of the ddeunadvise statement if you no longer want to be informed of a particular data item. The *ddechannel* is the channel ID returned from a successful ddeinitiate statement. The *itemname* is the data item about which you no longer want to be informed.

Use the second form of the ddeunadvise statement if you no longer want to be informed of changes to any data item for which ddeadvise requests have been made.

Versions of Crosstalk for Windows older than 2.0 do not support this statement.

Examples

```
ddeunadvise excelID, "R4C5"
```

In this example, a ddeunadvise request is sent, using the channel excelID, for the item R4C5.

```
ddeunadvise excelID
```

In this example, you request that all ddeadvise requests be canceled for the channel excelID.

B MACINTOSH CONSIDERATIONS

Writing scripts for a Macintosh environment B-2

Writing scripts for a Macintosh environment

When you write scripts to run in a Macintosh environment, keep in mind that Apple events allow other applications to communicate information to your script. The application that sends an Apple event is known as a source application, and the application receiving the event is called a target application.

With this version of CASL, a session can receive an event that requests it to run a script.

C

Error Return Codes

CASL error messages C-2

CASL error messages

Table C-1 lists the CASL error messages grouped by error class. The error class value is returned in the `errclass` system variable.

Table C-1. CASL error class values

Error class	Description
12	Compiler errors.
13	Input/output errors.
14	Mathematical and range errors.
15	State errors.
16	Critical errors.
17	Script execution errors.
18	Compatibility errors.
19	DOS gateway errors.
20	Call failure errors.
21	Missing information errors.
32	DDE errors.
42	Communications device errors.
44	Terminal errors.
45	File transfer errors.

Win

Win

The corresponding error codes for each class are listed in the following sections. For additional information about CASL errors, refer to the on-line help. Note that on-line error messages contain the most current information.

Compiler errors

Compiler errors are returned by the script compiler when your script is compiled. For an up-to-date list of these errors, refer to the on-line help.

Input/output errors

Input/output errors are explained in Table C-2. The error number is returned in the system variable `errno`.

Table C-2. Input/output errors

Error class and number	Explanation
13-01	Reserved.
13-02	An upload was canceled by the local operator.
13-03	Reserved.
13-04	A backup file cannot be created. There is insufficient room on the disk to receive the current file and also keep a backup copy.
13-05	The file number is invalid or missing.
13-06	The specified file channel number is already open. You must first close the channel or use another one.
13-07	The specified file channel number is not open.
13-08	Crosstalk cannot read an output file.
13-09	Crosstalk cannot write to an input file.
13-10	Crosstalk cannot `get/put` a text file.
13-11	Crosstalk cannot read from or write to a random file.
13-12	The file cannot be found in the `dirfil` path.
13-13	Reserved.
13-14	Reserved.
13-15	Reserved.
13-16	Window coordinates are out of range.
13-17	Reserved.
13-18	The specified window is not open.

continued

Table C-2. Input/output errors (cont.)

Error class and number	Explanation
13-19	Reserved.
13-20	Reserved.
13-21	Reserved.
13-22	Reserved.
13-23	Reserved.
13-24	Reserved.
13-25	Reserved.
13-26	This is an internal error. Contact DCA Technical Support.
13-27	Reserved.
13-28	An attempt to send output to the display failed.
13-29	A file copy failed.
13-30	The script attempted a `seek` in a sequential file; you can use `seek` only with random files.
13-31	Multiple windows in a session are not supported in this version.

Mathematical and range errors

Mathematical and range errors are explained in Table C-3. The error number is returned in the system variable `errno`.

Table C-3. Mathematical and range errors

Error class and number	Explanation
14-01	Arithmetic overflow has occurred.
14-02	Arithmetic underflow has occurred.
14-03	Division by zero was attempted.
14-04	The function key is out of range.

continued

Table C-3. Mathematical and range errors (cont.)

Error class and number	Explanation
14-05	The expression is not valid for the variable.
14-06	The value is outside the permissible range.
14-07	The value must be `on` or `off`.
14-08	Reserved.
14-09	A string was truncated.
14-10	Invalid characters were found in a numeric string.
14-11	The specified value is outside the acceptable range.
14-12	Reserved.
14-13	Reserved.
14-14	Reserved.
14-15	Reserved.
14-16	Reserved.
14-17	Reserved.
14-18	An invalid string was specified for the `quote` function.

State errors

State errors are explained in Table C-4. The error number is returned in the system variable `errno`.

Table C-4. State errors

Error class and number	Explanation
15-01	The specified command is applicable only when you are on line.
15-02	Reserved.
15-03	Reserved.

continued

Table C-4. State errors (cont.)

Error class and number	Explanation
15-04	Reserved.
15-05	Reserved.
15-06	Reserved.
15-07	The specified session does not currently exist.

Critical errors

Critical errors are explained in Table C-5. The error number is returned in the system variable `errno`.

Table C-5. Critical errors

Error class and number	Explanation
16-01	The device is write-protected.
16-02	The unit is unknown.
16-03	The drive is not ready.
16-04	The command is unknown.
16-05	A data error has occurred.
16-06	The request structure length is invalid.
16-07	A `seek` error has occurred.
16-08	The media type is unknown.
16-09	The sector cannot be found.
16-10	The printer is out of paper.
16-11	A write fault has occurred.
16-12	A read fault has occurred.
16-13	A general failure has occurred.
16-14	An open fault has occurred.
16-15	There is not enough memory available.

Script execution errors

Script execution errors are explained in Table C-6. The error number is returned in the system variable `errno`.

Table C-6. Script execution errors

Error class and number	Explanation
17-01	The specified label cannot be found.
17-02	Reserved.
17-03	`gosub` statements are nested too deep.
17-04	Reserved.
17-05	A data type mismatch for an external variable was found.
17-06	Reserved.
17-07	The script was canceled by the user.
17-08	A reference to an unresolved external variable was found.
17-09	Reserved.
17-10	An unavailable module variable was found.
17-11	Reserved.
17-12	A `return` statement without a corresponding `gosub` statement was found.
17-13	Reserved.
17-14	A script compilation failed when a `chain`, `do`, or `compile` statement was executed.
17-15	A return value was missing in the return from a function.
17-16	Reserved.
17-17	An internal error occurred. Delete the .xwc file and recompile the script. If the failure continues, contact DCA Technical Support.
17-18	An invalid `count` expression was used.
17-19	A string expression is too long.

continued

Table C-6. Script execution errors (cont.)

Error class and number	Explanation
17-20	There is not enough memory available.
17-21	A `dialog` item was used outside a `dialogbox/endialog` construct.
17-22	`dialogbox` statements are nested. These statements cannot be nested.
17-23	The dialog box cannot be displayed.
17-24	No pushbutton was specified for a dialog box.
17-25	A second `watch` statement was encountered before the first one was resolved.
17-26	Too many track channels are open.
17-27	A stack overflow has occurred. Procedures or functions are nested too deep.
17-28	The specified QuickPad file cannot be found.
17-29	The specified QuickPad has not been loaded.

Compatibility errors

Compatibility errors are explained in Table C-7. The error number is returned in the system variable `errno`.

Table C-7. Compatibility errors

Error class and number	Explanation
18-01	Reserved.
18-02	Reserved.
18-03	The .xwc file is bad. Recompile the .xws file.
18-04	Reserved.
18-05	The specified feature is not supported in this version.

Win DOS gateway errors

DOS gateway errors are explained in Table C-8. The error number is returned in the system variable `errno`.

Table C-8. DOS gateway errors

Error class and number	Explanation
19-01	An unexpected DOS error has occurred.
19-02	The specified file cannot be found.
19-03	The specified path cannot be found.
19-04	There are too many open files.
19-05	Access has been denied to the specified file.
19-06	The specified directory cannot be removed.
19-07	The diskette is write protected.
19-08	The disk is full.
19-09	There are invalid characters in the file name.
19-10	Reserved.
19-11	Reserved.
19-12	Reserved.
19-13	An invalid file name was specified.

■

Call failure errors

Call failure errors are explained in Table C-9. The error number is returned in the system variable `errno`.

Table C-9. Call failure errors

Error class and number	Explanation
20-01	The call was canceled by the user.
20-02	The modem did not detect the carrier when the call was answered or the call was never answered.

continued

Table C-9. Call failure errors (cont.)

Error class and number	Explanation
20-03	No dial tone was detected. The modem is set to check for dial tone before dialing and did not get a dial tone when it went off hook.
20-04	The number was busy. The modem detected a busy signal and was unable to make a connection.
20-05	A voice answer was detected.
20-06	There is no phone number for the connection. Choose Connection from the Settings pull-down to specify a number.
20-07	The connection is already in progress. Crosstalk was commanded to initiate a connection when one is already active.
20-08	The connection with the host has been terminated. This message is generated when Crosstalk disconnects from a host as a result of a user Disconnect request, when the call is terminated because the host disconnected the call, or when the call is dropped because of a connection failure.
20-09	A modem error has occurred. The modem returned an error indicating that it did not understand a command. Choose File Transfer from the Settings pull-down to check the modem command strings.
20-10	The modem did not respond. Crosstalk is not receiving a response from the modem after sending it a command. Choose File Transfer from the Settings pull-down to check the modem command strings.

Missing information errors

Missing information errors are explained in Table C-10. The error number is returned in the system variable `errno`.

Table C-10. Missing information errors

Error class and number	Explanation
21-01	The specified script file cannot be found. Check the name, make sure the file is in the DIRXWP directory, and try again.
21-02	The specified session file cannot be found. Check the name, make sure the file is in the DIRXWP directory, and try again.
21-03	The specified variable cannot be found.
21-04	The default file name is empty.
21-05	A file name argument is required but was omitted.
21-06	The format of the XWP directory is invalid. The session you attempted to start is from a version of Crosstalk that uses a different XWP file format.
21-07	Reserved.
21-08	Reserved.
21-09	There is no default file name; `filefind` must be used to set up a default file.

Win DDE errors

DDE errors are explained in Table C-11. The error number is returned in the system variable `errno`.

Table C-11. DDE errors

Error class and number	Explanation
32-01	The DDE channel number is invalid or missing. Review the syntax of the DDE statement and correct the channel number.
32-02	A bad response code from a PostMessage was returned internally by DDE. This can occur during periods of heavy system activity. Close the DDE connection and try again.
32-03	No response was received to a `ddeinitiate` request. Other applications are either busy or not in the system. Wait until another application is free or run a new copy of the application.
32-04	The data item about which you want to be advised is busy.
32-05	A `ddeunadvise` request was issued for an item that was not requested using `ddeadvise`.
32-06	An unknown data format was returned from the other application. Check the DDE documentation for the other application to determine other data retrieval methods.
32-07	A busy status was returned from the other application. Wait for the other application to finish and try the command again.
32-08	The command was rejected by the other application. This is normally caused by an invalid `ddeexecute` statement format. Review the DDE documentation for the other application to determine the correct format for the statement.

■

Communications device errors

Communications device errors are explained in Table C-12. The error number is returned in the system variable `errno`. Note that device errors are specific to the connection device you are using.

Table C-12. Communications device errors—direct connection

Error class and number	Explanation
42-01	The port is already in use.
42-02	The necessary hardware is not present.
42-03	The port is not open.
42-04	There is not enough memory for the communications buffers.
42-05	The specified serial port is not supported.
42-06	The specified baud rate is not supported.
42-07	The specified DataBits value is invalid.

Terminal errors

Terminal errors are explained in Table C-13. The error number is returned in the system variable `errno`.

Table C-13. Terminal errors

Error class and number	Explanation
44-01	An invalid terminal was selected.
44-02	An invalid terminal parameter was specified.

File transfer errors

File transfer errors are explained in Table C-14. The error number is returned in the system variable `errno`.

Table C-14. File transfer errors

Error class and number	Explanation
45-01	A general time-out has occurred.
45-02	The host is not responding. Check to make sure the communications link is working properly and try the transfer again.
45-03	An incorrect response from the host was received. The host computer did not respond as expected to your file transfer request. Check to make sure the communications link is working properly and try the transfer again.
45-04	Too many errors have occurred; the transfer is canceled. The transfer is automatically canceled because the maximum number of errors was reached. If the connection is noisy, try disconnecting and calling again. If the problem persists, change the protocol timing or raise the number of errors allowed before terminating.
45-05	The transfer was canceled because the connection was lost. Attempt to connect again and restart the transfer.
45-06	The transfer was canceled because of a sequencing failure. The protocol encountered an internal error. Try the transfer again. If the problem persists, contact DCA Technical Support.
45-07	The transfer was canceled by the local operator.
45-08	The transfer was canceled by the host computer.
45-09	A wild-card transfer was specified when using a protocol that cannot support wild-card specifications for the file name. Transfer a single file at a time or use a protocol that allows wild-card specifications.

continued

Table C-14. File transfer errors (cont.)

Error class and number	Explanation
45-10	The file to be transferred could not be found. The file name may be incorrect or the file may reside in a different directory.
45-11	The file transfer cannot take place or was canceled because the local disk is full.
45-12	The file transfer cannot take place or was canceled because the host disk is full.
45-13	The protocol has no server commands.
45-14	A file name is required for the transfer.
45-15	The system is busy. The system is performing tasks that prevent starting a file transfer. Wait for the task to finish and try the transfer again.
45-16	The protocol selected is not supported by Crosstalk.
45-17	The specified file transfer parameter is invalid.

D PRODUCT SUPPORT

Requesting technical support D-2

Accessing DCA on-line services D-3

Updating or upgrading your software D-3

Requesting technical support

If you encounter a problem installing or using Crosstalk and cannot find the answer in the documentation, you can call DCA Technical Support for help. Assistance is provided only to registered users. To register for technical support, complete the product registration card, which accompanies the software, and mail it to the following address:

DCA, Inc.
1000 Alderman Drive
Alpharetta, GA 30202-4199

Before contacting DCA Technical Support, make sure you know the following information:

- Your Crosstalk serial number. This number is on the master diskette.
- The version number of Crosstalk that you are using.
- The contents of your system files.

If possible, call the customer support department from a telephone that is near the PC you are using, so you can look at the software while working through the problem with DCA Technical Support.

You can call DCA Technical Support at (404) 442-3210. Representatives are available Monday through Friday, from 8:30 AM to 8:00 PM EST. You can also contact DCA Technical Support by FAX at (404) 442-4358.

Note: The telephone system at DCA Technical Support automatically routes calls to the next available representative, in the order in which the calls are received. Remain on the line until your call is answered to keep your place. ■

Accessing DCA on-line services

In addition to telephone support, DCA maintains a bulletin board service and a forum on CompuServe. These services provide the latest support files for all products, sample scripts, and technical assistance from DCA engineers.

The bulletin board service and the CompuServe forum can be accessed 24 hours a day, seven days a week. For instructions on how to connect to these services, refer to your Crosstalk user's guide.

Updating or upgrading your software

If you have any questions or concerns about disk updating, software versions, or compatibility issues that are not covered in this guide, contact DCA Technical Support.

Index

A

abs function, 6-3
absolute file paths, 2-8
accessing DCA on-line services, D-3
activatesession statement, 6-5
activate statement, 6-4
active function, 6-6
activesession function, 6-7, 6-237
Addition (operator), 1-20, 2-18, 2-19
add statement, 6-8, 6-226
alarm, sounding, 1-26
Alarm sounds (table), 6-9
alarm statement, 1-26, 6-9
alert statement, 1-20, 4-8, 6-11
and (operator), 2-22, 6-135
append mode, 6-289, 6-291
append option for the open statement, 6-190
Apple Comm ToolBox, 7-3, 7-5
Apple Modem Tool, 6-68, 7-4
Apple Serial Tool, 6-68, 7-4
arg function, 6-13, 6-79
arguments, passing to other scripts, 4-9
arithmetic expression, 1-20
  standard arithmetic operators, 2-18
arithmetic operators
  Addition, 2-18, 2-19
  BitAnd, 2-18, 2-19
  BitNot, 2-18, 2-19
arithmetic operators (cont.)
  BitOr, 2-18, 2-19
  BitXor, 2-18, 2-19
  Division, 2-18, 2-20
  IntDivision, 2-18, 2-20
  Modulo, 2-19, 2-20
  Multiplication, 2-18, 2-20
  Negate, 2-18, 2-20
  Rol, 2-18, 2-20
  Ror, 2-18, 2-20
  Shl, 2-18, 2-20
  Shr, 2-18, 2-20
  Subtraction, 2-18, 2-21
array declarations
  multidimensional, 3-7
  multidimensional, with alternative bounds, 3-8
  single-dimension, 3-7
  single dimension, with alternative bounds, 3-8
asc function, 6-15
ASCII control characters (table), 2-15
ASCII values in string constants, 2-14
assume statement, 6-16, 7-3, 7-4, 7-5

B

backups module variable, 6-17
basic CASL elements. *See* CASL elements, basic
binary function, 6-18
binary integers, 2-13

BitAnd (operator), 2-18, 2-19
Bitmap values
  for the chmod statement (table), 6-33
  for the fileattr function (table), 6-100
  for the fncheck function (table), 6-106
  for the fnstrip function (table), 6-107
  for the hms function (table), 6-133
BitNot (operator), 2-18, 2-19
BitOr (operator), 2-18, 2-19
bitstrip function, 6-19
BitXor (operator), 2-18, 2-19
blankex system variable, 6-20
blank lines, using, 1-14
block comments, 1-12, 2-3
bol option for the clear statement, 6-39
boolean
  constants, 2-16
  data type, 2-11
  expressions, 1-18, 2-22
boolean expression, testing an outcome with, 1-18
boolean operators
  and, 2-22
  not, 2-22
  or, 2-22
bow option for the clear statement, 6-39
braces
  to indicate a series of statements, 2-2
  using with a statement group, 1-21
branching to a different script location, 1-19
breaklen module variable, 6-21, 6-236
bulletin board service, DCA, xxi, D-3
bye session topic command, A-4
bye statement, 1-20, 6-22
byte data type, 2-11

C

call failure errors, C-9
calling another script, 4-9
call statement, 6-23, 6-156
cancel keyword, 6-11
cancel option, 6-75
cancel session topic command, A-4
capchars function, 6-24
capfile function, 6-25, 6-26
Capture options (table), 6-26
capture statement, 4-4, 6-25, 6-26
capture statement options
  new, 6-26
  off, 6-27
  on, 6-27
  pause, 6-27
  slash (/), 6-27
  to, 6-26
  toggle, 6-27
capture and upload control, 5-2
  add statement, 6-8
  blankex system variable, 6-20
  capchars function, 6-24
  capfile function, 6-25
  capture statement, 6-26
  cmode system variable, 6-42
  cwait statement, 6-57
  dirfil system variable, 6-77
  downloaddir system variable, 6-82
  grab statement, 6-126
  linedelim system variable, 6-154
  linetime system variable, 6-155

capture and upload control (cont.)
lwait statement, 6-162
tabex system variable, 6-252
upload statement, 6-272
capturing data, 4-4
case/endcase statement, 6-29
CASL
declarations, 3-2
errclass values, C-2
predeclared variables, 1-14
writing scripts, 1-6
CASL, rules for using
comments, 2-3
line continuation characters, 2-2
notational conventions, 2-4
statements, 2-2
CASL elements
changed, 8-3
new, 8-2
removed, 8-3
CASL elements, basic
compiler directives, 2-26
constants, 2-12
data types, 2-10
expressions, 2-17
general rules, 2-2
identifiers, 2-10
reserved keywords, 2-27
type conversion, 2-24
chaining to another script, 4-9
chain statement, 4-9, 6-13, 6-31, 6-79
character string, waiting for, 1-18, 4-2
char data type, 2-11
chdir statement, 6-32
child script, 3-6, 4-9
chmod statement, 6-33
choice system variable, 6-11, 6-35, 6-72
chr function, 2-25, 6-36
cksum function, 6-37, 6-48
class function, 6-38, 6-247
Class groupings (table), 6-38
clear statement, 6-39
clear statement options
bol, 6-39
bow, 6-39
eol, 6-39
eow, 6-39
line, 6-39
window, 6-39
close statement, 6-40
cls statement. *See* clear statement
cmode system variable, 6-42
cmode system variable options
normal, 6-42
raw, 6-42
visual, 6-42
Commands for the kermit statement (table), 6-148
comments
block, 2-3
line, 2-3
using in a script, 1-14, 2-3
Comm ToolBox, 7-3
communications device errors, C-13
communications device types, 6-68
compatibility errors, C-8
compatibility issues
Crosstalk for Macintosh, 8-4
Crosstalk Mark 4, 6-4, 6-5, 6-6, 6-12, 6-23, 6-165, 6-169, 8-4
Crosstalk for Windows, 8-2
compiler directives
genlabels, 2-26, 6-119
genlines, 2-26, 6-120
include, 2-27, 6-138
scriptdesc, 2-27, 6-232
trap, 2-26, 4-11, 6-267
compiler errors, C-3
compile statement, 6-43

compiling a script, 1-29
CompuServe forum, D-3
Conditions for the track statement (table), 6-262
Conditions for the wait statement (table), 6-277
Conditions for the watch statement (table), 6-280
connected function. *See* online function
Connection devices (table), 6-68
connection tool, 6-16, 6-68, 7-4
connectreliable module variable, 6-45
constants
  boolean, 2-16
  false, 6-99
  integer, 2-12
  off, 6-186
  on, 6-187
  real, 2-13
  string, 2-14
  true, 6-268
controlling the entire logon process, 1-22
controlling a process with a relational expression, 1-17
conventions
  documentation, xix
  DOS/Macintosh script file name, 2-8
  DOS/Macintosh terminology, 2-7
  DOS/Macintosh naming, 2-7
  notational, 2-4
conversions, type
  asc function, 6-15
  binary function, 6-18
  bitstrip function, 6-19
  chr function, 6-36
  class function, 6-38
  dehex function, 6-62
  detext function, 6-67
conversions, type (cont.)
  enhex function, 6-85
  entext function, 6-87
  hex function, 6-129
  intval function, 6-146
  mkint function, 6-172
  mkstr function, 6-173
  octal function, 6-185
  str function, 6-246
  val function, 6-274
converting
  an ASCII value to a character string, 2-25
  an integer to a hexadecimal string, 2-25
  an integer to a string, 2-24
  a string to an integer, 2-24
copy statement, 6-46
counters, incrementing, 1-20
count condition
  for the wait statement, 6-277
  for the watch/endwatch statement, 6-280
count function, 6-47
count option for the lwait statement, 6-162
crc function, 6-48
creating scripts with Learn, 1-5
critical errors, C-6
Crosstalk commands, executing using DDE, A-3
Crosstalk information, requesting using DDE, A-2
Crosstalk Mark 4, 6-6, 6-12, 6-23, 8-4
curday function, 6-49
curdir function, 6-50
curdrive function, 6-51
curhour function, 6-52
curminute function, 6-53
curmonth function, 6-54

cursecond function, 6-55, 6-258
curyear function, 6-56
cwait statement, 6-57
cwait statement options
  delay, 6-57
  echo, 6-57
  none, 6-57

D

data capture
  add statement, 6-8
  capchars function, 6-24
  capfile function, 6-25
  capture statement, 6-26
  cmode system variable, 6-42
  dirfil system variable, 6-77
  downloaddir system variable, 6-82
  grab statement, 6-126
data type conversion, 2-24
data types
  array, 2-11
  boolean, 2-11
  byte, 2-11
  char, 2-11
  integer, 2-11
  real, 2-11
  string, 2-11
  word, 2-11
date function, 6-59
date operations, 5-3
  curday function, 6-49
  curmonth function, 6-54
  curyear function, 6-56
  date function, 6-59
  weekday function, 6-282
DCAANSI emulation, 6-254
DCA ANSIPC Tool, 6-254
DCA Connection bulletin board, xxi
DCA CServeB Tool, 6-207
DCACSERV protocol, 6-207
DCADART protocol, 6-208
DCA DART Tool, 6-208
DCADEC emulation, 6-254
DCA DEC font, 7-3
DCA DEC Tool, 6-254
DCA FTTERM Tool, 6-255
DCAFTTRM emulation, 6-254
DCAHP700 emulation, 6-255
DCA HP700/94 Tool, 6-255
DCAIBM emulation, 6-255
DCA IBM3101 Tool, 6-255
DCA IND$FILE Tool, 6-208
DCAIND protocol, 6-208
DCAINT14 device, 6-68
DCA KERMIT Tool, 6-207
DCAKERMT protocol, 6-207
DCAMODEM device, 6-68
DCANASI device, 6-68
DCA on-line services, accessing, D-3
DCASERIL device, 6-68
DCA Technical Support, D-2
DCATTY emulation, 6-254
DCA TTY Tool, 6-254
DCA VIDTEX Tool, 6-254
DCAVIDTX emulation, 6-254
DCAWYSE emulation, 6-255
DCA WYSE Tool, 6-255
DCAXTALK protocol, 6-208
DCAXYMDM protocol, 6-207
DCA XYMODEM Tool, 6-207
DCAZMDM protocol, 6-207
DCA ZMODEM Tool, 6-207
ddeack statement, A-8
ddeadvisedatahandler event handler, A-10
ddeadvise statement, A-9
DDE commands, format of, A-12
DDE errors, C-12
ddeexecute statement, A-12
ddeinitiate statement, A-13

DDE interface, 5-4
command format, A-12
executing Crosstalk commands, A-3
requesting Crosstalk information, A-2
running demonstration scripts, A-6
session topic commands, A-4
session topic requests, A-3
system topic commands, A-4
system topic requests, A-2
topic name support, A-2
ddenak statement, A-15
ddepoke statement, A-16
dderequest statement, A-17
ddestatus function, A-18
ddeterminate statement, A-19
ddeunadvise statement, A-20
decimal integers, 2-12
declarations
arrays, 3-7
explicit, 3-4
func/endfunc, 6-116
functions, 3-12, 6-116
implicit, 3-5
procedures, 3-9, 6-204
proc/endproc, 6-204
public and external variables, 3-6
scope rules for labels, 3-15
scope rules for variables, 3-14
variables, 3-3
declaring variables in a script, 1-16
default keyword, 6-29
default variable initialization values, 3-14
defining a script description, 2-27
definput system variable, 6-60, 6-121, 6-212, 6-213
defoutput system variable, 6-61, 6-158, 6-289, 6-291
dehex function, 6-62, 6-85, 6-87
delay option
for the cwait statement, 6-57
for the lwait statement, 6-163
delete function, 6-64
delete statement, 6-63
delimiters, end of line, 2-9
demonstration scripts
provided for DDE, A-6
running, A-6
describing the purpose of a script, 1-12
description system variable, 6-65, 6-109, 6-128
designing a script, 1-10
destore function, 6-66, 6-86
detext function, 6-62, 6-67, 6-85, 6-87
developing a sample script
alerting the user if the connection failed, 1-20
branching to a different script location, 1-19
checking if a time-out occurred, 1-18
continuing the logon if the connection is established, 1-19
controlling the entire logon process, 1-22
declaring variables, 1-16
describing the purpose of the script, 1-12
disconnecting the session, 1-20
displaying a message, 1-12
documenting the script's history, 1-12
ending the script, 1-14
establishing a connection with MCI Mail, 1-13
incrementing a counter, 1-20

developing a sample script (cont.)
initializing variables, 1-17
logging on in a trouble-free environment, 1-11
overview, 1-11
performing a task while a condition is true, 1-17
performing a task while multiple conditions are true, 1-24
sending the logon sequence, 1-13
sounding an alarm, 1-26
testing an outcome with a boolean expression, 1-18
using braces with a statement group, 1-21
using CASL predeclared variables, 1-14
using comments and blank lines, 1-14
using indentation, 1-21
using keywords, 1-14
using the line-continuation sequence, 1-27
using a relational expression to control a process, 1-17
using string constants, 1-13
verifying the MCI Mail connection, 1-15
waiting for a character string, 1-18
waiting for a prompt from the host, 1-13
watching for one of several host responses, 1-24
device interaction, 5-5
connectreliable module variable, 6-45
dialmodifier module variable, 6-70
device system variable, 6-68, 7-4
device types, 6-68
dialmodifier module variable, 6-70
dialogbox/enddialog statement, 4-8, 6-71
dialog item options for the dialogbox/enddialog statement, 6-74
dialog items for the dialogbox/enddialog statement, 6-72
dial session topic command, A-4
dirfil system variable, 6-26, 6-77, 6-82, 6-214
disconnecting a session, 1-20
displaying information for a user, 4-6
displaying a message, 1-12
display system variable, 6-78, 6-201
Division (operator), 2-18, 2-20
documenting a script's history, 1-12
DOS gateway errors, C-9
DOS/Macintosh differences
absolute file paths, 2-8
end-of-line delimiters, 2-9
file path specifications, 2-8
naming conventions, 2-7
relative file paths, 2-8
script file name conventions, 2-8
terminology, 2-7
wild cards, 2-9
DOS and Macintosh terminology (table), 2-7
do statement, 4-9, 4-10, 6-13, 6-79
dosversion function, 6-81

double hyphens, to indicate a line comment, 2-3
downloaddir system variable, 6-26, 6-82, 6-214
drive statement, 6-83

E

echo option
  for the cwait statement, 6-57
  for the lwait statement, 6-162
emulations, 6-254
enabling error trapping, 4-11
ending a script, 1-14
end-of-line delimiters, 2-9
end statement, 1-14, 6-84
enhex function, 6-62, 6-85, 6-87
enstore function, 6-66, 6-86
entext function, 6-62, 6-67, 6-85, 6-87
environ function, 6-88
eof function, 6-89, 6-91
eol function, 6-91
eol option for the clear statement, 6-39
eow option for the clear statement, 6-39
Equality (operator), 2-22
errclass system variable, 4-11, 6-93, 6-94, 6-95, 6-267
errno system variable, 4-12, 6-94, 6-95, 6-267
error control, 5-5
  errclass system variable, 6-93
  errno system variable, 6-94
  error function, 6-95
  trap compiler directive, 6-267
error function, 4-11, 6-95, 6-267
error number, checking, 4-12
errors
  call failure, C-9
  communications device, C-13
  compatibility, C-8
  compiler, C-3
  critical, C-6
  DDE, C-12
  DOS gateway, C-9
  file transfer, C-14
  input/output, C-3
  mathematical and range, C-4
  missing information, C-11
  script execution, C-7
  state, C-5
  terminal, C-13
error trapping, 2-26, 4-11
error type, checking, 4-11
event handler, ddeadvisedatahandler, A-10
Excel, A-5
exchanging variables with other scripts, 4-10
executable file, 1-28
execute session topic command, A-5
exists function, 6-96
exit statement, 6-97, 6-204
explicit variable declarations
  multiple-variable declaration, 3-5
  single-variable declaration, 3-4
expressions
  arithmetic, 2-18
  boolean, 2-22
  relational, 2-21
  string, 2-21
expressions, order of evaluation, 2-17
external variables, 3-6, 4-10, 6-79
extract function, 6-98

F

false constant, 6-99, 6-268
fileattr function, 6-100
filedate function, 6-59, 6-102
filefind function, 6-100, 6-102, 6-103, 6-104, 6-105
file I/O operations, 5-5
  backups module variable, 6-17
  capture statement, 6-26
  chdir statement, 6-32
  chmod statement, 6-33
  close statement, 6-40
  copy statement, 6-46
  curdir function, 6-50
  curdrive function, 6-51
  definput system variable, 6-60
  defoutput system variable, 6-61
  delete statement, 6-63
  drive statement, 6-83
  eof function, 6-89
  eol function, 6-91
  exists function, 6-96
  fileattr function, 6-100
  filedate function, 6-102
  filefind function, 6-103
  filesize function, 6-104
  filetime function, 6-105
  fncheck function, 6-106
  fnstrip function, 6-107
  freefile function, 6-113
  get statement, 6-121
  kermit statement, 6-148
  loc function, 6-158
  mkdir statement, 6-171
  open statement, 6-190
  put statement, 6-209
  read line statement, 6-213
  read statement, 6-212
  receive statement, 6-214
  rename statement, 6-218
  rmdir statement, 6-228
file I/O operations (cont.)
  seek statement, 6-234
  send statement, 6-235
  upload statement, 6-272
  write line statement, 6-291
  write statement, 6-289
file paths
  absolute, 2-8
  relative, 2-8
file path specifications, 2-8
files, source and executable, 1-28
filesize function, 6-104
filetime function, 6-105
file transfer, 7-5
file transfer errors, C-14
File transfer protocols (table), 6-207
file transfer protocol types, 6-207
file transfer tool, 6-16, 6-207, 7-5
finish command for the kermit statement, 6-148
fncheck function, 6-106
fnstrip function, 6-107
fonts, provided with Crosstalk, 7-3
footer system variable, 6-109
formats system topic request, A-2
for/next statement, 6-110
forward declarations
  functions, 3-13, 6-116
  procedures, 3-10, 6-205
freefile function, 6-60, 6-113
freemem function, 6-114
freetrack function, 6-115, 6-261
func/endfunc declaration, 3-12, 6-116, 6-224
function declarations
  argument list, 3-12, 6-116
  forward function declaration, 3-13, 6-116

function declarations (cont.)
general description, 3-12, 6-116
variable and label references, 6-116
functions
abs, 6-3
active, 6-6
activesession, 6-7
arg, 6-13
asc, 6-15
binary, 6-18
bitstrip, 6-19
capchars, 6-24
capfile, 6-25
chr, 6-36
cksum, 6-37
class, 6-38
count, 6-47
crc, 6-48
curday, 6-49
curdir, 6-50
curdrive, 6-51
curhour, 6-52
curminute, 6-53
curmonth, 6-54
cursecond, 6-55
curyear, 6-56
date, 6-59
ddestatus, A-18
dehex, 6-62
delete, 6-64
destore, 6-66
detext, 6-67
dosversion, 6-81
enhex, 6-85
enstore, 6-86
entext, 6-87
environ, 6-88
eof, 6-89
eol, 6-91
error, 6-95
exists, 6-96
extract, 6-98
functions (cont.)
fileattr, 6-100
filedate, 6-102
filefind, 6-103
filesize, 6-104
filetime, 6-105
fncheck, 6-106
fnstrip, 6-107
freefile, 6-113
freemem, 6-114
freetrack, 6-115
hex, 6-129
hms, 6-133
inject, 6-139
inkey, 6-140
inscript, 6-143
insert, 6-144
instr, 6-145
intval, 6-146
left, 6-152
length, 6-153
loc, 6-158
lowcase, 6-159
max, 6-165
mid, 6-168
min, 6-169
mkint, 6-172
mkstr, 6-173
name, 6-175
nextchar, 6-178
nextline, 6-181
null, 6-183
octal, 6-185
online, 6-188
ontime, 6-189
pack, 6-192
pad, 6-193
quote, 6-211
right, 6-227
secno, 6-233
session, 6-237
sessname, 6-238
sessno, 6-239
slice, 6-244

functions (cont.)
str, 6-246
strip, 6-247
stroke, 6-249
subst, 6-250
systime, 6-251
time, 6-258
track, 6-265
upcase, 6-271
val, 6-274
version, 6-275
weekday, 6-282
winchar, 6-284
winsizex, 6-285
winsizey, 6-286
winstring, 6-287
winversion, 6-288
xpos, 6-292
ypos, 6-293
functions, declaring, 3-12
functions, external, 3-13

G

genlabels compiler directive, 2-26, 6-119, 6-143
genlines compiler directive, 2-26, 6-120
get command for the kermit statement, 6-148
get statement, 6-60, 6-89, 6-121, 6-234
go session topic command, A-5
go statement, 6-122
gosub/return statement, 6-116, 6-123, 6-198, 6-204
goto statement, 1-19, 6-116, 6-125, 6-146, 6-204
grab statement, 6-126
GreaterOrEqual (operator), 2-22
GreaterThan (operator), 2-22

H

halt statement, 6-127
header system variable, 6-128
help, on-line, xxi
hexadecimal integers, 2-12
hex function, 2-25, 6-129
hideallquickpads statement, 6-131
hidequickpad statement, 6-132
hide statement, 6-130, 6-240
hms function, 6-133
host computer, definition of, 1-3
host interaction, 5-7
breaklen module variable, 6-21
display system variable, 6-78
match system variable, 6-164
nextchar function, 6-178
nextline function, 6-181
nextline statement, 6-179
online function, 6-188
press statement, 6-199
reply statement, 6-221
sendbreak statement, 6-236
host prompt, waiting for, 1-13
host responses, watching for one of several, 1-24
hyphens (double) to indicate a line comment, 2-3

I

IBM-PC font, 7-3
identifiers, 2-10
if/then/else statement, 1-18, 6-135, 6-261, 6-265
implicit variable declarations, 3-5
include compiler directive, 2-27, 3-11, 3-13, 6-138, 6-205

including an external file, 2-27
incrementing a counter, 1-20
indentation, using, 1-21
Inequality (operator), 2-22
initializing variables, 1-17
inject function, 6-139
inkey function, 6-140, 6-178, 6-199, 6-249
input mode, 6-60, 6-212, 6-213
input option for the open statement, 6-190
input/output errors, C-3
input statement, 4-7, 6-142
inscript function, 6-143
insert function, 6-144
instr function, 6-47, 6-145
IntDivision (operator), 2-18, 2-20
integer data type, 1-16, 2-11
Integer ranges for the octal function (table), 6-185
integers
  binary, 2-13
  decimal, 2-12
  hexadecimal, 2-12
  kilo, 2-13
  octal, 2-13
Integer values and their binary string lengths (table), 6-18
interacting with the host, 4-2
intval function, 2-24, 6-146, 6-246, 6-274
invoking other scripts, 4-9

J

jump statement. *See* goto statement

K

kermit statement, 6-148
kermit statement commands
  finish, 6-148
kermit statement commands (cont.)
  get, 6-148
  send, 6-148
Keyboard keys and their corresponding numbers (table), 6-140
key names in string constants, 2-16
key condition
  for the track statement, 6-263
  for the wait statement, 6-277
  for the watch/endwatch statement, 6-280
keys system variable, 6-150
keywords, 1-9, 1-14, 2-27
kilo integers, 2-13

L

labels
  overview, 1-9
  scope rules, 3-15
label statement, 1-19, 6-151
Learn facility
  recording a script, 1-5
  replaying a script, 1-6
left function, 6-152
length function, 6-153
LessOrEqual (operator), 2-22
LessThan (operator), 2-22
line comments
  using double hyphens, 1-12, 2-3
  using a semicolon, 2-4
line continuation characters, 1-27, 2-2
linedelim system variable, 6-154
line option for the clear statement, 6-39
linetime system variable, 6-155, 6-163

loadallquickpads statement, 6-156
loadquickpad statement, 6-157
load session topic command, A-5
load statement, 6-156
load system topic command, A-4
loc function, 6-158, 6-234
logon, continuing if a connection is established, 1-19
logon sequence, sending to the host, 1-13
lowcase function, 6-159
lprint statement, 6-160
lwait statement, 6-57, 6-155, 6-162
lwait statement options
  count, 6-162
  delay, 6-163
  echo, 6-162
  none, 6-162
  prompt, 6-162

M

Macintosh connections, supported, 7-4
Macintosh environments, writing scripts for, B-2
Macintosh/DOS differences
  absolute file paths, 2-8
  end-of-line delimiters, 2-9
  file path specifications, 2-8
  naming conventions, 2-7
  relative file paths, 2-8
  script file name conventions, 2-8
  terminology, 2-7
  wild cards, 2-9
match system variable, 6-164, 6-263
mathematical operations, 5-8
  abs function, 6-3
  cksum function, 6-37
  crc function, 6-48
  intval function, 6-146
  max function, 6-165
  min function, 6-169
  mkint function, 6-172
  val function, 6-274
mathematical and range errors, C-4
max function, 6-165
maximize statement, 6-166
MCI Mail connection
  establishing, 1-13
  verifying, 1-15
message statement, 1-12, 1-20, 4-7, 6-167
Microsoft Excel, A-5
mid function, 6-168
min function, 6-169
minimize statement, 6-170
missing information errors, C-11
mkdir statement, 6-171
mkint function, 6-172, 6-173
mkstr function, 6-172, 6-173
Mode options for the open statement (table), 6-190
modifying
  connection tool variables, 7-4
  file transfer tool variables, 7-5
  terminal tool variables, 7-3
module variables
  backups, 6-17
  breaklen, 6-21
  connectreliable, 6-45
  dialmodifier, 6-70
  number, 6-184
  patience, 6-196
  redialcount, 6-216
  redialwait, 6-217
  tabwidth, 6-253

Modulo (operator), 2-18, 2-20
move statement, 6-174
Multiplication (operator), 2-18, 2-20

N

name function, 6-175
Negate (operator), 2-18, 2-20
netid system variable, 6-176
new option for the capture statement, 6-26
new session topic command, A-5
new statement, 6-177
new system topic command, A-4
nextchar function, 6-178
nextline function, 6-181
nextline statement, 1-18, 6-179, 6-259
noask keyword, 6-63
none option
  for the cwait statement, 6-57
  for the lwait statement, 6-162
normal option for the cmode statement, 6-42
notational conventions
  angle brackets, 2-5
  bold square brackets, 2-6
  bold braces, 2-6
  ellipses, 2-6
  typeface, 2-5
not (operator), 2-22, 6-135
null function, 6-183
number module variable, 6-184

O

octal function, 6-185
octal integers, 2-13
off constant, 6-186, 6-187
off option for the capture statement, 6-27
ok keyword, 6-11
ok option for the dialogbox/enddialog statement, 6-75
on constant, 6-186, 6-187
online function, 1-24, 6-188
on-line help, xxi
on option for the capture statement, 6-27
ontime function, 6-189
open statement, 6-190
open statement options
  append, 6-190
  input, 6-190
  output, 6-190
  random, 6-190
Options for the clear statement (table), 6-39
Options for the cmode variable (table), 6-42
Options for the cwait statement (table), 6-57
or (operator), 2-22, 6-135
output mode, 6-61, 6-289, 6-291
output option for the open statement, 6-190

P

pack function, 6-192
pad function, 6-193, 6-209
Parameters for the lwait statement (table), 6-162
parent script, 3-6
passing arguments to other scripts, 4-9
password system variable, 1-14, 6-195
path specifications, 2-8
patience module variable, 6-196

pause option for the capture statement, 6-27
perform statement, 3-11, 6-197, 6-205
pop statement, 6-198
predefined variables
  module, 3-3
  system, 3-3
  using, 1-14
press statement, 6-199
printer control, 5-8
  footer system variable, 6-109
  header system variable, 6-128
  lprint statement, 6-160
  printer system variable, 6-203
printer system variable, 6-203
print statement, 4-6, 6-201
procedure declarations
  argument list, 3-9
  forward declarations, 3-10, 6-205
  general description, 3-9, 6-204
  variable and label references, 6-204
procedures
  declaring, 3-9
  external, 3-11
proc/endproc declaration, 3-9, 6-204
program flow control
  case/endcase statement, 6-29
  chain statement, 6-31
  do statement, 6-79
  end statement, 6-84
  exit statement, 6-97
  for/next statement, 6-110
  freetrack function, 6-115
  func/endfunc declaration, 6-116
  gosub/return statement, 6-123
  goto statement, 6-125
  halt statement, 6-127
  if/then/else statement, 6-135
program flow control (cont.)
  label statement, 6-151
  new statement, 6-177
  perform statement, 6-197
  proc/endproc declaration, 6-204
  quit statement, 6-210
  repeat/until statement, 6-219
  return statement, 6-224
  terminate statement, 6-257
  timeout system variable, 6-259
  trace statement, 6-260
  track function, 6-265
  track statement, 6-261
  wait statement, 6-276
  watch/endwatch statement, 6-279
  while/wend statement, 6-283
prompt option for the lwait statement, 6-162
protocol system variable, 6-207, 7-5
protocol types, 6-207
public variables, 3-6, 4-10, 6-79
public variable session topic request, A-3
pull-down, definition of, xx
pushbutton accelerator, 6-12, 6-72
put statement, 6-61, 6-209, 6-234

Q

quiet condition
  for the track statement, 6-263
  for the wait statement, 6-277
  for the watch/endwatch statement, 6-280
quit statement, 6-210
quotation marks embedded in string constants, 2-14
quote function, 6-211, 6-289

R

random mode, 6-60, 6-61, 6-121, 6-209, 6-234
random option for the open statement, 6-190
Range of coordinates for the move statement (table), 6-174
range and mathematical errors, C-4
raw option for the cmode statement, 6-42
read line statement, 6-60, 6-89, 6-213
read statement, 6-60, 6-89, 6-91, 6-212
real constants, 2-13
real data type, 2-11
receive statement, 6-214
recorded scripts
  recording with Learn, 1-5
  replaying, 1-6
redialcount module variable, 6-184, 6-196, 6-216, 6-217
redialwait module variable, 6-196, 6-217
relational expressions
  boolean comparisons, 2-21
  using in a script, 1-17
relational operators
  Equality, 2-22
  GreaterOrEqual, 2-22
  GreaterThan, 2-22
  Inequality, 2-22
  LessOrEqual, 2-22
  LessThan, 2-22
relative file paths
  DOS, 2-8
  Macintosh, 2-9
rename statement, 6-218
repeat/until statement, 4-3, 6-219, 6-283
replaying a script, 1-6
reply statement, 1-13, 4-6, 6-86, 6-154, 6-199, 6-221
requesting Crosstalk information using DDE, A-2
requesting information from a user, 4-7
request statement. *See* receive statement
reserved keywords, 2-27
restore statement, 6-223, 6-240
return codes, C-2
return statement, 6-123, 6-224
rewind statement, 6-226
right function, 6-227
rmdir statement, 6-228
Rol (operator), 2-18, 2-20
Ror (operator), 2-18, 2-20
run statement, 6-229

S

sample scripts. *See also* developing a sample script
  controlling the entire logon process, 1-22
  developing, 1-11
  logging on in a trouble-free environment, 1-11
  verifying the MCI Mail connection, 1-15
saveas session topic command, A-5
save session topic command, A-5
save statement, 6-230
scope rules
  for global variables, 3-14
  for labels, 3-15
  for local variables, 3-14
scriptdesc compiler directive, 2-27, 6-232

Script description, defining, 2-27
script elements
  constants, 1-9
  expressions, 1-9
  keywords, 1-9
  labels, 1-9
  procedures and functions, 1-9
  statements, 1-8
  variables, 1-9
script execution errors, C-7
script file name conventions, 2-8
script file types, 1-28
script management, 5-10
  chain statement, 6-31
  compile statement, 6-43
  do statement, 6-79
  genlabels compiler directive, 6-119
  genlines compiler directive, 6-120
  include compiler directive, 6-138
  inscript function, 6-143
  quit statement, 6-210
  scriptdesc compiler directive, 6-232
  startup system variable, 6-245
  terminate statement, 6-257
  trace statement, 6-260
scripts
  calling another, 4-9
  chaining to another, 4-9
  compiling, 1-29
  designing, 1-10
  developing a sample, 1-11
  ending one, 1-14
  exchanging variables with other scripts, 4-10
  invoking, 4-9
  passing arguments to other scripts, 4-9
  recording with Learn, 1-5
scripts (cont.)
  replaying a recorded script, 1-6
  running, 1-30
  script elements, 1-8
  script structure, 1-7
  script types, 1-6
  why use them, 1-4
  writing for a Macintosh environment, B-2
script structure
  comments, 1-7
  declarations, 1-7
  directives, 1-8
script system variable, 6-231
secno function, 6-233
seek statement, 6-60, 6-234
semicolon to indicate a line comment, 2-4
sendbreak statement, 6-236
send command for the kermit statement, 6-148
sending a logon sequence, 1-13
sending a reply to the host, 4-6
send statement, 6-235
Serial Tool, 6-68, 7-4
session, disconnecting, 1-20
session function, 6-237
session management, 5-10
  activatesession statement, 6-5
  active function, 6-6
  activesession function, 6-7
  assume statement, 6-16
  bye statement, 6-22
  call statement, 6-23
  description system variable, 6-65
  device system variable, 6-68
  dirfil system variable, 6-77
  downloaddir system variable, 6-82
  go statement, 6-122
  keys system variable, 6-150
  load statement, 6-156

session management (cont.)
name function, 6-175
netid system variable, 6-176
number module variable, 6-184
ontime function, 6-189
password system variable, 6-195
patience module variable, 6-196
protocol system variable, 6-207
quit statement, 6-210
redialcount module variable, 6-216
redialwait module variable, 6-217
run statement, 6-229
save statement, 6-230
session function, 6-237
sessname function, 6-238
sessno function, 6-239
startup system variable, 6-245
terminal system variable, 6-254
terminate statement, 6-257
userid system variable, 6-273
session topic commands
bye, A-4
cancel, A-4
dial, A-4
execute, A-5
go, A-5
load, A-5
new, A-5
save, A-5
saveas, A-5
session topic requests
public variable, A-3
status, A-3
sessname function, 6-238
sessno function, 6-239
setting and testing time limits, 4-5
Shl (operator), 2-18, 2-20
showallquickpads statement, 6-241
showquickpad statement, 6-242
show statement, 6-240
Shr (operator), 2-18, 2-20
Single-dimension arrays, 3-7
size statement, 6-243
slash (/) option for the capture statement, 6-27
slice function, 6-244
software, updating or upgrading, D-3
some keyword, 6-46
sounding an alarm, 1-26
source file, 1-28
special characters in string constants, 2-16
startup system variable, 6-245
state errors, C-5
statement group, using braces with, 1-21
statements
activatesession, 6-5
add, 6-8
alarm, 6-9
alert, 6-11
assume, 6-16
bye, 6-22
call, 6-23
capture, 6-26
case/endcase, 6-29
chain, 6-31
chdir, 6-32
chmod, 6-33
clear, 6-39
close, 6-40
compile, 6-43
copy, 6-46
cwait, 6-57
ddeack, A-8
ddeadvise, A-9
ddeexecute, A-12
ddeinitiate, A-13

statements (cont.)
ddenak, A-15
ddepoke, A-16
dderequest, A-17
ddeterminate, A-19
ddeunadvise, A-20
delete, 6-63
dialogbox/enddialog, 6-71
do, 6-79
drive, 6-83
end, 6-84
exit, 6-97
for/next, 6-110
get, 6-121
go, 6-122
gosub/return, 6-123
goto, 6-125
grab, 6-126
halt, 6-127
hide, 6-130
hideallquickpads, 6-131
hidequickpad, 6-132
if/then/else, 6-135
input, 6-142
kermit, 6-148
label, 6-151
load, 6-156
loadallquickpads, 6-156
loadquickpad, 6-157
lprint, 6-160
lwait, 6-162
maximize, 6-166
message, 6-167
minimize, 6-170
mkdir, 6-171
move, 6-174
new, 6-177
nextline, 6-179
open, 6-190
perform, 6-197
pop, 6-198
press, 6-199
print, 6-201
statements (cont.)
put, 6-209
quit, 6-210
read, 6-212
read line, 6-213
receive, 6-214
rename, 6-218
repeat/until, 6-219
reply, 6-221
restore, 6-223
return, 6-224
rewind, 6-226
rmdir, 6-228
run, 6-229
save, 6-230
seek, 6-234
send, 6-235
sendbreak, 6-236
show, 6-240
showallquickpads, 6-241
showquickpad, 6-242
size, 6-243
terminate, 6-257
trace, 6-260
track, 6-261
unloadallquickpads, 6-269
unloadquickpad, 6-270
upload, 6-272
wait, 6-276
watch/endwatch, 6-279
while/wend, 6-283
write, 6-289
write line, 6-291
zoom, 6-294
status session topic request, A-3
status system requests, A-2
str function, 2-24, 6-246
string (case) condition
for the track statement, 6-262
for the wait statement, 6-277
for the watch/endwatch statement, 6-280

string constants
ASCII values, 2-14
continuing on a new line, 2-16
embedded quotation marks, 2-14
general description, 2-14
key names, 2-16
special characters, 2-16
unprintable characters, 2-14
using in a script, 1-13
string data type, 2-11
string expressions, 2-21
string operations, 5-13
arg function, 6-13
bitstrip function, 6-19
count function, 6-47
dehex function, 6-62
delete function, 6-64
destore function, 6-66
detext function, 6-67
enhex function, 6-85
enstore function, 6-86
entext function, 6-87
extract function, 6-98
hex function, 6-129
hms function, 6-133
inject function, 6-139
insert function, 6-144
instr function, 6-145
intval function, 6-146
left function, 6-152
length function, 6-153
lowcase function, 6-159
mid function, 6-168
mkstr function, 6-173
null function, 6-183
pack function, 6-192
pad function, 6-193
quote function, 6-211
right function, 6-227
slice function, 6-244
str function, 6-246
strip function, 6-247
string operations (cont.)
subst function, 6-250
upcase function, 6-271
val function, 6-274
winstring function, 6-287
string (space) condition
for the track statement, 6-262
for the wait statement, 6-277
for the watch/endwatch statement, 6-280
strip function, 6-98, 6-193, 6-247
stroke function, 6-249
subst function, 6-250
Subtraction (operator), 2-18, 2-21
support, technical, D-2
supported Windows connections, 7-4
suppressing label information, 2-26
suppressing line number information, 2-26
sysitems system topic request, A-2
system topic commands
load, A-4
new, A-4
system topic requests
formats, A-2
status, A-2
sysitems, A-2
topics, A-2
system variables
blankex, 6-20
choice, 6-35
cmode, 6-42
definput, 6-60
defoutput, 6-61
description, 6-65
device, 6-68
dirfil, 6-77
display, 6-78
downloaddir, 6-82

system variables (cont.)
errclass, 6-93
errno, 6-94
footer, 6-109
header, 6-128
keys, 6-150
linedelim, 6-154
linetime, 6-155
match, 6-164
netid, 6-176
password, 6-195
printer, 6-203
protocol, 6-207
script, 6-231
startup, 6-245
tabex, 6-252
terminal, 6-254
timeout, 6-259
userid, 6-273
systime function, 6-251

T

tabex system variable, 6-252
tabstop group option for the dialogbox/enddialog statement, 6-74
tabstop option for the dialogbox/enddialog statement, 6-74
tabwidth module variable, 6-252, 6-253
takes keyword, 6-204
technical support, D-2
Terminal emulations (table), 6-254
terminal errors, C-13
terminal system variable, 6-254, 7-3
terminal tool, 6-16, 6-254, 7-3
terminal types, 6-254
terminate statement, 6-257
testing if an error occurred, 4-11
testing an outcome with a boolean expression, 1-18
tick, definition of, 1-25
tick keyword, 6-189, 6-251
time function, 6-105, 6-258
time operations, 5-3
curhour function, 6-52
curminute function, 6-53
cursecond function, 6-55
hms function, 6-133
secno function, 6-233
time function, 6-258
time-out, checking if one occurred, 1-18
timeout system variable, 4-5, 6-179, 6-259, 6-277
toggle option for the capture statement, 6-27
tool concept, 7-2
tools
connection, 6-16, 6-68, 7-4
file transfer, 6-16, 6-207, 7-5
terminal, 6-16, 6-254, 7-3
to option for the capture statement, 6-26
topic commands
session, A-4
system, A-4
topic name support for DDE, A-2
topic requests
session, A-3
system, A-2
trace statement, 6-260
track clear, 6-263
track function, 6-265
track routine, 6-261, 6-263, 6-265
track statement, 6-115, 6-261
track statement conditions
key, 6-263
quiet, 6-263
string (case), 6-262
string (space), 6-262

trap compiler directive, 2-26, 4-11, 6-95, 6-267
trapping an error, 2-26, 4-11
true constant, 6-99, 6-268
type conversion, 5-14
  asc function, 6-15
  binary function, 6-18
  bitstrip function, 6-19
  chr function, 6-36
  class function, 6-38
  dehex function, 6-62
  detext function, 6-67
  enhex function, 6-85
  entext function, 6-87
  hex function, 6-129
  intval function, 6-146
  mkint function, 6-172
  mkstr function, 6-173
  octal function, 6-185
  str function, 6-246
  val function, 6-274

U

unloadallquickpads statement, 6-269
unloadquickpad statement, 6-270
unprintable characters in string constants, 2-14
upcase function, 6-271
updating software, D-3
upload control, 5-2
  blankex system variable, 6-20
  cwait statement, 6-57
  linedelim system variable, 6-154
  linetime system variable, 6-155
  lwait statement, 6-162
  tabex system variable, 6-252
  upload statement, 6-272
upload statement, 6-272
user
  communicating with, 4-6
  requesting information from, 4-7
userid system variable, 1-14, 6-273

V

val function, 6-274
variable declarations
  explicit, 3-4
  implicit, 3-5
  public and external, 3-6
variables
  backups module variable, 6-17
  blankex system variable, 6-20
  breaklen module variable, 6-21
  choice system variable, 6-35
  cmode system variable, 6-42
  connectreliable module variable, 6-45
  default initialization values, 3-14
  definput system variable, 6-60
  defoutput system variable, 6-61
  description system variable, 6-65
  device system variable, 6-68
  dialmodifier module variable, 6-70
  dirfil system variable, 6-77
  display system variable, 6-78
  errclass system variable, 6-93
  errno system variable, 6-94
  exchanging with other scripts, 4-10
  external, 3-6, 4-10
  footer system variable, 6-109
  global, 3-14

variables (cont.)
global, default initialization values, 3-14
header system variable, 6-128
initializing, 1-17, 3-6
keys system variable, 6-150
local, 3-14
local, default initialization values, 3-14
match system variable, 6-164
netid system variable, 6-176
number module variable, 6-184
password system variable, 6-195
patience module variable, 6-196
predefined, 3-3
printer system variable, 6-203
protocol system variable, 6-207
public, 3-6, 4-10
redialcount module variable, 6-216
redialwait module variable, 6-217
scope rules, 3-14
script system variable, 6-231
startup system variable, 6-245
tabex system variable, 6-252
tabwidth module variable, 6-253
terminal system variable, 6-254
timeout system variable, 6-259
user-defined, 3-4
userid system variable, 6-273
verifying the MCI Mail connection, 1-15
version function, 6-275
visual option for the cmode statement, 6-42

W

waiting for a character string, 1-18, 4-2
waiting for a prompt from the host, 1-13
wait statement, 1-13, 1-18, 4-2, 6-276
wait statement conditions
count, 6-277
key, 6-277
quiet, 6-277
string (case), 6-277
string (space), 6-277
watch/endwatch statement, 1-24, 4-3, 6-279
watch/endwatch statement conditions
count, 6-280
key, 6-280
quiet, 6-280
string (case), 6-280
string (space), 6-280
watching for one of several events to occur, 4-3
watching for one of several host responses, 1-24
weekday function, 6-282
while/wend statement, 1-17, 1-24, 4-3, 6-219, 6-280, 6-283
wild-card support, 2-9
winchar function, 6-284
window control, 5-15
activate statement, 6-4
alert statement, 6-11
choice system variable, 6-35
clear statement, 6-39
dialogbox/enddialog statement, 6-71
hideallquickpads statement, 6-131
hidequickpad statement, 6-132
hide statement, 6-130

window control (cont.)
input statement, 6-142
loadallquickpads statement, 6-156
loadquickpad statement, 6-157
maximize statement, 6-166
message statement, 6-167
minimize statement, 6-170
move statement, 6-174
print statement, 6-201
restore statement, 6-223
showallquickpads statement, 6-241
showquickpad statement, 6-242
show statement, 6-240
size statement, 6-243
tabwidth module variable, 6-253
unloadallquickpads statement, 6-269
unloadquickpad statement, 6-270
winchar function, 6-284
winsizex function, 6-285
winsizey function, 6-286
winstring function, 6-287
xpos function, 6-292
ypos function, 6-293
zoom statement, 6-294
window option for the clear statement, 6-39
Windows connections, supported, 7-4
winsizex function, 6-285
winsizey function, 6-286
winstring function, 6-287
winversion function, 6-288
word data type, 2-11
write line statement, 6-61, 6-291
write statement, 6-61, 6-289
writing scripts with CASL, 1-6

X

xpos function, 6-292
XTALK.INI file, 3-3

Y

ypos function, 6-293

Z

zoom statement, 6-294

DCA
1000 Alderman Drive
Alpharetta, GA 30202-4199

General Information: 404 442 4930
Technical Support: 404 442 3210

Part No. 016590