

# CASL Script Language Guide

Attachmate®  
**INFOCONNECT.**  
Enterprise Edition

**UNISYS**

## **Copyrights and Notices**

### **Attachmate® INFOConnect® Enterprise Edition**

© 2011 Attachmate Corporation. All Rights Reserved.

#### **Patents**

This Attachmate software is protected by U.S. patents 6252607 and 6803914.

#### **Trademarks**

Attachmate, the Attachmate logo, CryptoConnect, FileXpress, and PEPgate are either registered trademarks or trademarks of Attachmate Corporation in the USA. INFOConnect is a registered trademark of Unisys Corporation. FIPS 140-1 Validated is a certification mark of NIST, which does not imply product endorsement by NIST, the U.S. or Canadian Governments. All other trademarks, trade names, or company names referenced in product materials are used for identification only and are the property of their respective owners.

#### **Attachmate Software License Agreement**

A copy of the Attachmate software license agreement governing this product can be found in a 'license' file in the root directory of the product.

#### **Licensors**

Attachmate Corporation  
1500 Dexter Avenue North  
Seattle, WA 98109 USA  
USA  
+1.206.217.7100  
<http://www.attachmate.com>

#### **Third-Party Notices**

Third Party Terms and notices are provided in a 'thirdpartynotices' file in the root directory of the product.

---

# Contents

	<b>About This Guide</b>	<b>xv</b>
	Audience . . . . .	xvi
	Documentation Conventions . . . . .	xvii
	Abbreviations . . . . .	xx
	Related Documentation . . . . .	xxi
<b>Chapter 1</b>	<b>Introducing CASL</b>	<b>1</b>
	About CASL . . . . .	2
	Why Use Macros? . . . . .	3
	Creating and Editing CASL Macros . . . . .	4
	Creating a CASL Macro . . . . .	4
	Types of Macros . . . . .	6
	The Structure of Macros . . . . .	7
	Comments . . . . .	7
	Declarations . . . . .	7
	Directives . . . . .	8
	The Elements of a Macro . . . . .	9
	Statements . . . . .	9
	Variables . . . . .	9
	Constants . . . . .	9
	Expressions . . . . .	9
	Labels . . . . .	9
	Procedures and Functions . . . . .	9
	Keywords . . . . .	10

<b>Chapter 1</b>	<b>Introducing CASL, continued</b>	
	Designing a Macro . . . . .	11
	Sample: A Basic Logon Macro . . . . .	12
	Describing the Purpose of the Macro . . . . .	12
	Documenting the Macro's History . . . . .	13
	Displaying a Message . . . . .	13
	Using String Constants . . . . .	13
	Establishing Communications with MCI Mail . . . . .	14
	Waiting for a Prompt from the Host . . . . .	14
	Sending the Logon Sequence . . . . .	14
	Using CASL Predeclared Variables . . . . .	14
	Using Keywords . . . . .	15
	Ending the Macro . . . . .	15
	Using Comments and Blank Lines . . . . .	15
	Sample: Verifying the Host Connection . . . . .	16
	Declaring Variables . . . . .	17
	Initializing Variables . . . . .	18
	Performing a Task While a Condition Is True . . . . .	18
	Using a Relational Expression to Control the Process . . . . .	18
	Waiting for a Character String . . . . .	18
	Checking if a Timeout Occurred . . . . .	19
	Testing the Outcome with a Boolean Expression . . . . .	19
	Branching to a Different Macro Location . . . . .	19
	Continuing the Logon if the Connection Is Established . . . . .	20
	Incrementing a Counter Using an Arithmetic Expression . . . . .	20
	Alerting the User if the Connection Failed . . . . .	20
	Disconnecting the Session . . . . .	21
	Using Indentation . . . . .	21
	Using Braces with a Statement Group . . . . .	22
	Sample: Controlling the Entire Logon Process . . . . .	23
	Performing a Task while Multiple Conditions Are True . . . . .	25
	Watching for One of Several Host Responses . . . . .	25
	Sounding an Alarm . . . . .	27
	Using the Line-Continuation Sequence . . . . .	28
	Compiling a CASL Macro . . . . .	29
	Running a CASL Macro . . . . .	30
<b>Chapter 2</b>	<b>Understanding the Basics of CASL</b>	<b>31</b>
	Statements . . . . .	32
	Line Continuation Characters . . . . .	32
	Comments . . . . .	33
	Block Comments . . . . .	33

<b>Chapter 2</b>	<b>Understanding the Basics of CASL, continued</b>	<b>63</b>
	Line Comments . . . . .	33
	Identifiers . . . . .	35
	Data Types . . . . .	36
	Constants . . . . .	37
	Integer Constants . . . . .	37
	Real Constants . . . . .	38
	String Constants . . . . .	39
	Boolean Constants . . . . .	43
	Expressions . . . . .	44
	Order of Evaluation . . . . .	45
	Arithmetic Expressions . . . . .	46
	String Expressions . . . . .	50
	String Concatenation Operation . . . . .	50
	Relational Expressions . . . . .	51
	Boolean Expressions . . . . .	53
	Type Conversion . . . . .	54
	Converting an Integer to a String . . . . .	54
	Converting a String to an Integer . . . . .	54
	Converting an Integer to a Hexadecimal String . . . . .	54
	Converting an ASCII Value to a Character String . . . . .	55
	Compiler Directives . . . . .	56
	Suppressing Label Information . . . . .	56
	Suppressing Line Number Information . . . . .	56
	Trapping an Error . . . . .	56
	Including an External File . . . . .	57
	Defining a Macro Description . . . . .	57
	Reserved Keywords . . . . .	58
 <b>Chapter 3</b>	 <b>Variables, Arrays, Procedures, and Functions</b>	 <b>63</b>
	Variables . . . . .	64
	Predefined Variables . . . . .	64
	User-Defined Variables . . . . .	64
	Explicit Variable Declarations . . . . .	65
	Single-Variable Declarations . . . . .	65
	Multiple-Variable Declarations . . . . .	65
	Initializers . . . . .	66
	Public and External Variables . . . . .	66
	Implicit Variable Declarations . . . . .	67
	Arrays . . . . .	68
	Single-Dimensional Arrays . . . . .	68
	Multidimensional Arrays . . . . .	68

<b>Chapter 3</b>	<b>Variables, Arrays, Procedures, and Functions, continued</b>	
	Arrays with Alternative Bounds . . . . .	69
	Procedures. . . . .	70
	Procedure Argument Lists. . . . .	70
	Forward Declarations for Procedures . . . . .	71
	External Procedures . . . . .	72
	Functions . . . . .	73
	Function Argument Lists . . . . .	73
	Forward Declarations for Functions. . . . .	74
	External Functions. . . . .	74
	Scope Rules. . . . .	75
	Local Variables . . . . .	75
	Global Variables . . . . .	75
	Default Variable Initialization Values . . . . .	75
	Labels . . . . .	76
	Calling DLL Functions . . . . .	77
	Declaring DLL Functions. . . . .	77
	Parameter and Return Values . . . . .	78
	Non-Supported Parameters and Return Values . . . . .	80
	Writing Windows DLLs . . . . .	80
<b>Chapter 4</b>	<b>Interacting with the Host, Users, and Other Macros</b>	<b>83</b>
	Interacting with the Host . . . . .	84
	Waiting for a Character String . . . . .	84
	Watching for Conditions to Occur . . . . .	85
	Setting and Testing Time Limits . . . . .	86
	Sending a Reply to the Host . . . . .	86
	Communicating with a User. . . . .	87
	Displaying Information. . . . .	87
	Requesting Information . . . . .	88
	Invoking Other Macros. . . . .	90
	Chaining to Another Macro . . . . .	90
	Calling Another Macro. . . . .	90
	Passing Arguments . . . . .	90
	Exchanging Variables . . . . .	91
	Trapping and Handling Errors . . . . .	92
	Enabling Error Trapping . . . . .	92
	Testing if an Error Occurred . . . . .	92
	Checking the Type of Error . . . . .	92
	Checking the Error Number. . . . .	92

<b>Chapter 5</b>	<b>Functional Purpose of CASL Elements</b>	<b>95</b>
	Overview . . . . .	96
	Date and Time Operations . . . . .	97
	Error Control . . . . .	98
	File Input/Output Operations . . . . .	99
	Host Interaction . . . . .	101
	Macro Management . . . . .	102
	Mathematical Operations . . . . .	103
	Printer Control . . . . .	104
	Program Flow Control . . . . .	105
	Session Management . . . . .	107
	String Operations . . . . .	109
	Type Conversion Operations . . . . .	111
	Window Control . . . . .	112
	Miscellaneous Elements . . . . .	114
 <b>Chapter 6</b>	 <b>CASL Language</b>	 <b>115</b>
	How CASL Elements Are Documented . . . . .	124
	abs (function) . . . . .	125
	activate (statement) . . . . .	126
	activatesession (statement) . . . . .	127
	alarm (statement) . . . . .	128
	alert (statement) . . . . .	130
	arg (function) . . . . .	132
	asc (function) . . . . .	133
	assume (statement) . . . . .	134
	backups (module variable) . . . . .	135
	binary (function) . . . . .	136
	bitstrip (function) . . . . .	137
	busycursor (statement) . . . . .	138
	bye (statement) . . . . .	139
	capture (statement) . . . . .	140
	case...endcase (statements) . . . . .	142
	chain (statement) . . . . .	144
	chdir (statement) . . . . .	145
	choice (system variable) . . . . .	146
	chr (function) . . . . .	147
	cksum (function) . . . . .	148
	class (function) . . . . .	149
	clear (statement) . . . . .	150
	close (statement) . . . . .	151
	cls (statement) . . . . .	152

**Chapter 6 CASL Language, continued**

compile (statement) . . . . .	153
connected (function) . . . . .	154
copy (statement) . . . . .	155
count (function) . . . . .	156
crc (function) . . . . .	157
curday (function) . . . . .	158
curdir (function) . . . . .	159
curdrive (function) . . . . .	160
curhour (function) . . . . .	161
curminute (function) . . . . .	162
curmonth (function) . . . . .	163
cursecond (function) . . . . .	164
curyear (function) . . . . .	165
date (function) . . . . .	166
definput (system variable) . . . . .	167
defoutput (system variable) . . . . .	168
dehex (function) . . . . .	169
delete (statement) . . . . .	170
delete (function) . . . . .	171
description (system variable) . . . . .	172
destore (function) . . . . .	173
detext (function) . . . . .	174
device (system variable) . . . . .	175
dialogbox...enddialog (statements) . . . . .	176
display (system variable) . . . . .	183
do (statement) . . . . .	184
drive (statement) . . . . .	186
end (statement) . . . . .	187
enhex (function) . . . . .	188
enstore (function) . . . . .	189
entext (function) . . . . .	190
environ (function) . . . . .	191
eof (function) . . . . .	192
eol (function) . . . . .	193
errclass (system variable) . . . . .	195
errno (system variable) . . . . .	196
error (function) . . . . .	197
exists (function) . . . . .	198
exit (statement) . . . . .	199
false (constant) . . . . .	200
filefind (function) . . . . .	201
filesize (function) . . . . .	203
fncheck (function) . . . . .	204



**Chapter 6 CASL Language, continued**

fnstrip (function) . . . . . 205

footer (system variable) . . . . . 207

for...next (statements) . . . . . 208

freemem (function) . . . . . 210

freetrack (function) . . . . . 211

func...endfunc (function declaration) . . . . . 212

genlabels (compiler directive) . . . . . 214

genlines (compiler directive) . . . . . 215

get (statement) . . . . . 216

go (statement) . . . . . 217

gosub...return (statements) . . . . . 218

goto (statement) . . . . . 219

grab (statement) . . . . . 220

halt (statement) . . . . . 221

header (system variable) . . . . . 222

hex (function) . . . . . 223

hide (statement) . . . . . 224

hideallquickpads (statement) . . . . . 225

hidequickpad (statement) . . . . . 226

hms (function) . . . . . 227

homedir (system variable) . . . . . 228

if...then...else (statements) . . . . . 229

include (compiler directive) . . . . . 231

inject (function) . . . . . 232

inkey (function) . . . . . 233

input (statement) . . . . . 235

inscript (function) . . . . . 236

insert (function) . . . . . 237

instr (function) . . . . . 238

intval (function) . . . . . 239

jump (statement) . . . . . 240

keys (system variable) . . . . . 241

label (statement) . . . . . 242

left (function) . . . . . 243

length (function) . . . . . 244

loadquickpad (statement) . . . . . 245

loc (function) . . . . . 246

lowercase (function) . . . . . 247

lprint (statement) . . . . . 248

match (system variable) . . . . . 249

max (function) . . . . . 250

maximize (statement) . . . . . 251

mid (function) . . . . . 252

**Chapter 6 CASL Language, continued**

min (function) . . . . .	253
minimize (statement) . . . . .	254
mkdir (statement) . . . . .	255
mkint (function) . . . . .	256
mkstr (function) . . . . .	257
move (statement) . . . . .	258
name (function) . . . . .	259
netid (system variable) . . . . .	260
new (statement) . . . . .	261
nextchar (function) . . . . .	262
nextline (statement) . . . . .	263
nextline (function) . . . . .	265
null (function) . . . . .	267
octal (function) . . . . .	268
off (constant) . . . . .	269
on (constant) . . . . .	270
online (function) . . . . .	271
ontime (function) . . . . .	272
open (statement) . . . . .	273
pack (function) . . . . .	274
pad (function) . . . . .	275
passchar (system variable) . . . . .	277
password (system variable) . . . . .	278
perform (statement) . . . . .	279
pop (statement) . . . . .	280
press (statement) . . . . .	281
print (statement) . . . . .	283
printer (system variable) . . . . .	284
proc...endproc (procedure declaration) . . . . .	285
protocol (system variable) . . . . .	288
put (statement) . . . . .	289
quit (statement) . . . . .	290
quote (function) . . . . .	291
read (statement) . . . . .	292
read line (statement) . . . . .	293
receive (statement) . . . . .	294
rename (statement) . . . . .	295
repeat...until (statements) . . . . .	296
reply (statement) . . . . .	297
request (statement) . . . . .	298
restore (statement) . . . . .	299
return (statement) . . . . .	300
right (function) . . . . .	301

**Chapter 6 CASL Language, continued**

rmdir (statement) . . . . .	302
run (statement) . . . . .	303
save (statement) . . . . .	304
script (system variable) . . . . .	305
scriptdesc (compiler directive) . . . . .	306
secno (function) . . . . .	307
seek (statement) . . . . .	308
send (statement) . . . . .	309
sendbreak (statement) . . . . .	310
session (function) . . . . .	311
sessname (function) . . . . .	312
sessno (function) . . . . .	313
show (statement) . . . . .	314
showallquickpads (statement) . . . . .	315
showquickpad (statement) . . . . .	316
size (statement) . . . . .	317
slice (function) . . . . .	318
startup (system variable) . . . . .	319
str (function) . . . . .	320
strip (function) . . . . .	321
stroke (function) . . . . .	322
subst (function) . . . . .	323
systemtime (function) . . . . .	324
tabwidth (module variable) . . . . .	325
terminal (system variable) . . . . .	326
terminate (statement) . . . . .	327
time (function) . . . . .	328
timeout (system variable) . . . . .	329
trace (statement) . . . . .	330
track (statement) . . . . .	331
track (function) . . . . .	334
trap (compiler directive) . . . . .	336
true (constant) . . . . .	337
unloadallquickpads (statement) . . . . .	338
unloadquickpad (statement) . . . . .	339
upcase (function) . . . . .	340
userid (system variable) . . . . .	341
val (function) . . . . .	342
version (function) . . . . .	343
wait (statement) . . . . .	344
watch...endwatch (statements) . . . . .	348
weekday (function) . . . . .	351
while...wend (statements) . . . . .	352

<b>Chapter 6</b>	<b>CASL Language, continued</b>	
	winchar (function) . . . . .	353
	winsize (function) . . . . .	354
	winsizey (function) . . . . .	355
	winstring (function) . . . . .	356
	winversion (function) . . . . .	357
	write (statement) . . . . .	358
	write line (statement) . . . . .	359
	xpos (function) . . . . .	360
	ypos (function) . . . . .	361
	zoom (statement) . . . . .	362
<b>Chapter 7</b>	<b>Connection, Terminal, and File Transfer Tools</b>	<b>363</b>
	Tool Concept . . . . .	364
	Connection Tools . . . . .	365
	Terminal Tools . . . . .	366
	File Transfer Tools . . . . .	367
	Using Tool Variables . . . . .	368
	Connection Tool Variables . . . . .	369
	IT 27 Variables . . . . .	370
	UTS Variables . . . . .	374
<b>Appendix A</b>	<b>Error Messages</b>	<b>379</b>
	Classes of Error Message . . . . .	380
	Internal Errors . . . . .	381
	Compiler Errors . . . . .	382
	Input/Output Errors . . . . .	390
	Mathematical and Range Errors . . . . .	392
	State Errors . . . . .	393
	Critical Errors . . . . .	394
	Macro Execution Errors . . . . .	395
	Compatibility Errors . . . . .	398
	Upload/Download Errors . . . . .	399
	Missing Information Errors . . . . .	400
	Multiple Document Interface Errors . . . . .	401

<b>Appendix A</b>	<b>Error Messages, continued</b>	
	Emulator or File Transfer Protocol Errors . . . . .	402
	DLL Errors . . . . .	403
	Generic Module Errors . . . . .	404
	File Transfer Errors . . . . .	405
	Navigation Errors . . . . .	407
	<b>Index</b>	<b>409</b>



---

## ***About This Guide***

The *INFOConnect CASL Script Language Guide* is designed to assist you in creating and implementing macros that enhance communication between your PC and host. It introduces CASL, the Common Accessory Script Language. This guide explains how to use CASL with Accessory Manager.

This preface contains the following sections:

<a href="#">Audience</a> .....	xiv
<a href="#">Documentation Conventions</a> .....	xv
<a href="#">Abbreviations</a> .....	xviii
<a href="#">Related Documentation</a> .....	xix

## **Audience**

This guide is written for Accessory Manager users who want to write CASL macros. It begins with conceptual information so that the inexperienced programmer can learn the hows and whys of writing macros. The guide provides reference material on implementing each macro element. This reference material also includes details for the sophisticated application developer.

If you are new to writing macros, you may benefit from first reading [Chapter 1, “Introducing CASL.”](#)

Before reading this guide, you should understand general concepts for Accessory Manager.



## Documentation Conventions

The following documentation conventions are used in this guide:

- All text that you type on a screen or messages and prompts that appear on the screen are displayed in this type style.

This type style also is used for CASL macro text.

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

```
alarm [integer]
```

In this example, the argument *integer* is optional.

- Words or characters in 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.

- An ellipsis ( `...` ) can have one of several meanings.
  - If the ellipses occurs at the end of a line, it indicates that the line is continued on the following line, or that the code continues but no additional data is shown, as in these examples:

```
[edittext x, y, w, h, init_text, ...
  str_result_var [, options]]
```

```
if arg(1) = "barkley" then ...
```

- If the ellipses occurs on a line of its own, it indicates that intervening lines of code have been omitted, as in the following example:

```
done = false
while not done
  ...
  ...
wend
```

- If the ellipses follows an item in italics, you can repeat the previous item one or more times, as in the following example:

*digit ...*

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

- If the ellipses follows an item in square brackets, you can repeat the item zero or more times, as in the following example:

[, var] ...

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

- Italic type is used in the following situations:
  - To show emphasis, as in, “Do *not* use the Copy command.”
  - To show that a word is a placeholder that stands for something else, as in the following example:

`delete filename`

In this case, you enter the actual file name rather than the word *filename*.

The following are some common placeholders:

*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 word *PC* refers to any personal computer running Windows
  
- The word *host* refers to any mainframe, mini-computer, or information hub with which the *PC* communicates.
  
- File names are shown in all capital letters, as in `INSTALL.EXE`, unless a file name is part of a command. In this situation, lowercase letters are used to show that you do not have to enter the file name in all capitals.

## Abbreviations

The following abbreviations are used in this guide.

<b>Abbreviation</b>	<b>Meaning</b>
API	Application programming interface
ASCII	American Standard Code for Information Interchange
BPS	Bits per second
CASL	Common Accessory Script Language
CR	Carriage return
CRC	Cyclical redundancy check
CR/LF	Carriage-return/line-feed
DLL	Dynamic Link Library
DTE	Data Terminal Equipment
FCC	Federal Communications Commission
GDI	Graphical Device Interface
IEEE	Institute of Electrical and Electronics Engineers
KB	Kilobyte

***About This Guide***

---



---

# *Introducing CASL*

# 1

## **In This Chapter**

This chapter contains the following headings:

About CASL . . . . .	2
Why Use Macros? . . . . .	3
Creating and Editing CASL Macros . . . . .	4
Types of Macros . . . . .	6
The Structure of Macros . . . . .	7
The Elements of a Macro . . . . .	9
Designing a Macro . . . . .	11
Sample: A Basic Logon Macro . . . . .	12
Sample: Verifying the Host Connection . . . . .	16
Sample: Controlling the Entire Logon Process . . . . .	23
Compiling a CASL Macro . . . . .	29
Running a CASL Macro . . . . .	30

## About CASL

CASL is a scripting language that you can use to create macros that can interact with hosts, users, and other macros. The macros you develop can be simple or complex. For instance, you can create a simple macro that waits for a prompt from the host and then replies with a user ID and password. More complex macros can automate entire communications sessions or create custom dialog boxes that enable users to operate a host application without learning its commands.

While CASL is designed to simplify the process of communicating with other computers, it is not limited to that function. CASL is a full-featured programming language that can handle almost any task, including complex mathematical computations and the display of sophisticated dialog boxes.

CASL macros work with any emulator that runs within Accessory Manager. Any limitations that are specific to a particular emulator (such as ALC or EXTRA!<sup>®</sup> Enterprise for Accessory Manager) are noted throughout this guide or the Readme file for the product.



## Why Use Macros?

When you work in a data communication environment, you often have to perform the same functions over and over again to complete your daily activities. For instance, each time you open a session with a host, you have to type your logon ID and password.

You can eliminate the manual repetition of routine tasks by using macros to communicate with the host. You have to create and save a macro to be able to use it, but once you have done this, you will find it invaluable in saving time and effort in the future.

Using macros, you can do any of the following:

- Perform keystroke sequences
- Run another PC application
- Perform almost any function that can be performed using Accessory Manager, such as loading a QuickPad™
- Create dialog boxes so that you can request user input

In addition, creating and implementing CASL macros are not difficult tasks. Traditionally, developing applications and utilities that run in a communications environment required a complex programming language and an Application Programming Interface (API) to access the host. You also had to understand the underlying data communications link. CASL removes these obstacles. When you write a CASL macro, you do not have to concern yourself with the details of communication programming; CASL handles the communication interface.

## Creating and Editing CASL Macros

### Creating a CASL Macro

You can create a CASL macro in two ways:

- Learn Mode—you perform the actions that you want to include in the macro, and Accessory Manager records those actions in a CASL macro file, which you can then edit if needed.
- CASL Macro Editor—you open the CASL Macro Editor and write the macro using the CASL script language.



**Caution:** CASL macro files have a size limit of 64K bytes.

### Using Learn Mode

To create a CASL macro using Learn Mode, follow these steps:

- 1 With a session open, click Learn CASL Macro from the Tools menu.  
  
The CASL Macro Editor starts in a minimized state.
- 2 Perform the tasks that you want to include in the macro.
- 3 When you have finished, click Stop CASL Learn from the Tools menu.
- 4 When you are prompted about saving the CASL macro, do one of the following:

To do this	Do this
Save the CASL macro	Click Yes, type a name for the macro in the File Name text box (you do not have to include a file extension), and click Save on the Save As dialog box.  Note: In INFOConnect 9.1 and later, the CASL Macro Editor encrypts macros (and any related source files) during the save process if the Encrypt macros setting is selected in the Accessory Manager Global Preferences dialog box.
Not save the CASL macro	Click No.

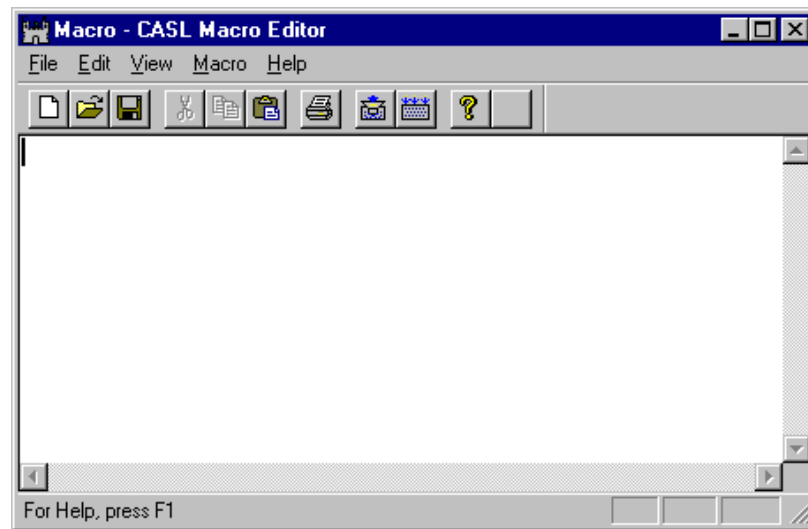
If you need to edit the CASL macro, you can do so using the CASL Macro Editor. Refer to the online Help for Accessory Manager for detailed information.

Using the CASL Macro Editor

To create a CASL macro using the CASL Macro Editor, follow these steps:

- 1 With a session open, click CASL Macro from the Tools menu.
- 2 Click New.

The CASL Macro Editor starts, displaying a window similar to the one shown below:



For information about using this editor, refer to the online Help.

## Types of Macros

There are two main types of CASL macros:

- Online
- Offline

Online macros work while Accessory Manager is connected to a host. Usually, these interact with the host to automate all or part of a communications session. You can use online macros to log on to the host, or create a custom dialog box for interacting with a host application.

Offline macros do not interact with a host. For example, you can use an offline macro to display a list of hosts to which a user might want to log on.

**Note:** A session must be open for you to run either an online or an offline macro.

## The Structure of Macros

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 macro include such items as comments, declarations, and directives. A comment documents a macro; a declaration defines a variable, an array, a procedure, or function; and a directive specifies 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 macro that you do not want to execute. Comments are ignored by the macro compiler and do not take up any memory after a macro is compiled. So you can include many comments to document the flow of a macro.

Starting your macro with a comment header is good practice. This header should include your name, the creation date, and some explanation of its objective. An example of this type of comment is as follows:

```
-- Macro name: LOGON.XWS
-- Date:    6/24/2000
-- Author:  John Doe
```

In this example, the double hyphen 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 after the comment header. This will help you keep the declarations easy to find, as shown here:

```
-- Macro name: LOGON.XWS
-- Date:    6/24/2000
-- Author:  John Doe
integer count, access_number
count = 1
access_number = NetID
```

## Directives

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

```
print "Hello!"
```

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

To make your macro 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.

## The Elements of a Macro

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

<b>Statements</b>	Statements perform such functions as assignment of values, file input/output, file transfer, macro flow control, host interaction, window control, and communications session management. CASL statements are described in detail in <a href="#">Chapter 6, “CASL Language.”</a>
<b>Variables</b>	Variables are elements that can store data. In your macros, you can use variables that you create and variables that are predeclared by CASL. CASL’s predeclared variables are described in <a href="#">Chapter 6, “CASL Language.”</a>
<b>Constants</b>	Constants are elements that have a fixed value. Use the value directly in your macro.
<b>Expressions</b>	Expressions include arithmetic expressions, string expressions, relational expressions, and Boolean expressions.
<b>Labels</b>	Labels are named reference points in a macro. A label can be the destination of a <code>goto</code> statement or it can mark the beginning of a subroutine. Guidelines for using the <code>label</code> statement in a macro are presented in <a href="#">Chapter 6, “CASL Language.”</a> Label scope rules are explained in <a href="#">Chapter 3, “Variables, Arrays, Procedures, and Functions.”</a>
<b>Procedures and Functions</b>	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 <a href="#">Chapter 6, “CASL Language,”</a> for details.

**Keywords**

Keywords make your macro more readable. CASL keywords are reserved for a particular use in your macro; 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.



## Designing a Macro

In the process of developing and implementing a more complex macro, the following is a typical development cycle:

- Design the macro.
- Write and edit the macro.
- Compile the macro and locate any compile errors.
- Fix the errors and compile again.
- Run the macro to be sure it works.
- Correct any problems.

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

Your macro 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.

## Sample: A Basic Logon Macro

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

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

-- Macro name: SAMPLE1.XWS
-- Created:    6/24/2000 - 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 macro
```

### Describing the Purpose of the Macro

The macro begins with a comment describing the purpose of the macro.

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

This is a block comment, which is enclosed in the symbol pair `/*` and `*/`. When you start your macro with an explanatory comment, you assist other macro writers who later need to understand your work.

### Documenting the Macro's History

The sample macro comment header also provides a history of the script's development, including the macro file name, the creation date, and the author's name. This comment begins with a double hyphen, which tells the macro compiler that this is a line comment. Line comments do not require an end-of-comment symbol.

```
-- Macro name: SAMPLE.XWS  
-- Created: 6/24/2000 - Jane Smith
```

After subsequent macro modifications, the header might appear as follows:

```
-- Macro name: SAMPLE.XWS  
-- Created: 2/14/2000 - Jane Smith  
-- Modified: 4/23/2000 - Jane Smith  
-- Modified: 5/10/2000 - John Doe
```

The additional comments record the history of the macro development.

### Displaying a Message

The first line of code displays a message that tells 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 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. 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 macro to pause for two seconds to allow the host to respond with the first prompt. The amount of time to wait depends 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 macro 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 macros and also to help maintain security. You can define these variables from Accessory Manager by clicking Session Preferences from the Options menu and clicking the CASL Macro tab. You can also modify these variables in a macro. The sample macro 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 macro more readable and flexible. Use the keywords only where specified in the various language elements.

**Ending the Macro**

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

The `end` statement brings the macro to an orderly conclusion. Other CASL statements, such as `halt`, `quit`, and `terminate`, cause related macros, sessions, or Accessory Manager to end also. These statements are discussed in detail in [Chapter 6, “CASL Language.”](#)

**Using Comments and Blank Lines**

Throughout the sample macro 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.

```
-- Macro name: SAMPLE.XWS  
reply      -- Send a CR
```

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

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

## Sample: Verifying the Host Connection

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

```
/* This macro 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. */

-- Macro name:SAMPLE2.XWS
-- Created:6/24/2000 - Jane Smith
-- Modified:6/25/2000 - 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-logout 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 macro
```

## Declaring Variables

As in the first sample macro, this sample starts with a description of its purpose and an outline of its history. (The comment header is updated to reflect a modification to the original macro.) This macro adds functionality that takes control in the event that MCI Mail does not respond to the initial reply statement.

First the macro 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 macro branches to a different location to send the logon sequence to the application. If, however, communications cannot be established after ten carriage returns are sent, the macro alerts the user to the failure, disconnects the session, and ends.

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

```
integer i
```

This macro uses only one variable. If your macro 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 macro compiler initializes an integer variable to a default value of 0. To initialize the variable to a different value, use the equal sign (=). In the sample macro, 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 macro 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 `<` and `=`) are called relational expressions. When you use these operators, the result is always a Boolean value (true or false). In this macro, 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 macro to wait for one specific text string, use the `wait` statement. This sample macro waits for the character string "port:" to ensure that a connection with MCI Mail is established. To prevent the macro from waiting forever, a duration time of two seconds is specified.

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

You can determine if a time-out occurred before the character string arrived, as explained in the next section.



**Checking if a Timeout 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 macro, `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 macro, 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 macro:

```
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 macro, if the Boolean expression is `true`, the macro transfers control to a `logon` routine, which is located in a different part of the macro, as explained in the next section.

**Branching to a Different Macro Location**

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

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

To enable the macro compiler to know where to branch, you must supply a label name in the goto statement. In the sample macro, 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`. For detailed information about this statement, refer to [“gosub...return \(statements\)”](#) on page 218.

### Continuing the Logon if the Connection Is Established

If the macro 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 macro increments the `while...wend` conditional 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 macro, 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 macro alerts the user to the failure.

### Alerting the User if the Connection Failed

In general, the sample macro uses the message statement to inform the user of current events. A message, which is displayed 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 command to exit the dialog box. In the sample macro, the `alert` statement provides an OK button for the user.

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

The macro pauses at the `alert` statement until the user clicks OK.

**Disconnecting the Session**

If the connection with MCI Mail cannot be established, the macro uses the `bye` statement to end the session. The `bye` statement immediately disconnects the current session.

**Using Indentation**

As you can see, some of the lines of code in the macro 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 macro more readable. If indentation was not used in the sample macro, 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 macro, 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
}
```

## Sample: Controlling the Entire Logon Process

In the previous examples, the sample macros did not verify the logon prompts sent by the host and therefore did not take corrective action if a prompt never appeared. In this macro, 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 macro gives you greater control over the sequence of events that may occur when communicating with your host.

```

/* This macro 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 and uses the watch
statement to verify that the logon sequence is
successfully sent to the host. */

-- Macro 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"

/* Send 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
-- Increment counter
i = i + 1
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 macro

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 macro, which verified the MCI Mail connection, this macro 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 macro uses a more comprehensive technique to ensure that it sends the correct logon responses to the host.

Based on two controlling conditions (the macro is online and tries is less than 5), the macro 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 macro responds appropriately to whichever host prompt or message it receives.

**Performing a Task while Multiple Conditions Are True**

In the previous sample macro, the `while...wend` construct contained one relational expression that determined how many times the while loop was repeated. This macro 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, macro execution continues with the statement following the `wend`.

The `online` function returns true as long as the macro is online to the host. 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 four times. It may be repeated fewer than four times, depending on what happens while the macro 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 macro watches ten seconds for six 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 ten-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 macro 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 macro, 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 two consecutive seconds.

```
quiet 2 seconds : reply
```

If this condition is met, the macro sends a carriage return to MCI Mail and processing returns to the `while...wend` construct. If the macro 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 macro receives the "name:" prompt, it waits five ticks (a tick is one tenth of a second) and then sends the contents of `userid` to MCI Mail. If the macro 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 macro executes the statements associated with the third watch condition.

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



After a brief wait of five ticks, the macro 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 macro receives this message, it waits five 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 macro 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 macro 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 macro displays a dialog box and waits for the user to click OK; then it ends.

**Sounding an Alarm**

To get the user's attention, you can use the alarm statement to make the PC emit a sound. This macro uses the alarm statement, with an argument of 1.

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

The `alarm` statement argument determines the type of sound that the PC makes. In this case, an argument of `1` specifies that the PC should play the `.WAV` file associated with the `SystemAsterisk` key in the Windows Registry. For more information about alarm sounds, refer to “[alarm \(statement\)](#)” on page 128.

### Using the Line-Continuation Sequence

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

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

**Note:** You can skip using the line continuation sequence and keep the entire statement on one line. However, the statement may be too long to fit in your editor window, and you will have to scroll to the right and left to see the entire line.

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 a CASL Macro

Once you have created and saved a CASL macro, you should compile it to determine possible syntax errors. The compiler converts your source macro 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 macro.

There are two types of macro files:

- Source file (.XWS), which you create and edit
- Executable file (.XWC), which is created when you compile your macro

### Procedure

To compile a CASL macro, follow these steps:

- 1 If the CASL macro that you want to compile is not already open, open it.

From an Accessory Manager session, click CASL Macro from the Tools menu, click the desired .XWS file, and click Edit.

From the CASL Macro Editor, click Open from the File menu and double-click the desired .XWS file.

- 2 From the Macro menu, click Compile.
- 3 If any compilation errors occur, correct the errors.
- 4 Repeat steps 2 and 3 until your macro compiles without errors.

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

## Running a CASL Macro

You can run macros at any of the following times:

- When you start Accessory Manager (application start-up macro)
- When you open a session (session start-up macro)
- When you click CASL Macro from the Tools menu, click the desired macro, and click Run
- When you click a toolbar or QuickPad button, press a key, or double-click a HotSpot that has been configured to run a macro
- When the left mouse double-click has been configured to run a macro with the same name as the word under the mouse pointer
- When you click Run from the CASL Macro Editor's Macro menu

For detailed information about these procedures, refer to the online Help for Accessory Manager.

---

# ***Understanding the Basics of CASL***

# **2**

## **In This Chapter**

This chapter includes the following headings:

Statements .....	32
Comments .....	33
Identifiers .....	35
Data Types .....	36
Constants .....	37
Expressions .....	44
Arithmetic Expressions .....	46
String Expressions .....	50
Relational Expressions .....	51
Boolean Expressions .....	53
Type Conversion .....	54
Compiler Directives .....	56
Reserved Keywords .....	58

## 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:

```
activate
```

- Multiple statements to a logical line with a colon (:) between each statement, as 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 macro 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 macro. Comments are useful for maintaining, modifying, or debugging the macro in the future.

You can add two types of comments to a macro:

- Block comments
- Line comments

### 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 macro 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 macro 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 macro processor sorts out the pairs correctly.

Be careful when using block comments, however. If you fail to terminate the block comment correctly, the compiler will treat every statement in the rest of the macro 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 (`--`)
- Semicolon (`;`)

**Note:** 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 in the middle of identifiers or string constants).

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

```
-- Macro name: HELLO.XWS  
-- Date: 12-18-2000
```

**Semicolon**

Use the semicolon indicator only in a location where you would normally place a CASL statement, as shown in the following examples:

```
print "Hi," : ; This is a comment  
  
reply userid  
; Send your user ID to the host
```



## Identifiers

Each variable, procedure, function, label, and other type of element used in a macro 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), using 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, “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:

Data Type	Description
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.

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

## 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:

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

### Decimal Integers

Decimal integers use a base of 10, which means that 0 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 through 9 and also a through f (lowercase 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	<p>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 through 7 are valid digits. The following are examples:</p> <pre>17o 17Q</pre>
Binary Integers	<p>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 macro 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.</p> <p>The following is an example of a binary constant:</p> <pre>1001001B</pre>
Kilo Integers	<p>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 macro processor encounters a k or K following an integer constant, it multiplies the constant by 1,024. For example, 32K becomes 32,768.</p> <p>The following are examples of kilo integers:</p> <pre>64K 128k</pre>
Real Constants	<p>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.</p> <p>Real constants have one of the following formats:</p> <pre>[-] [digit...] "." digit... [exponent] [-] digit... exponent</pre> <p>The <i>exponent</i> has the following format:</p> <pre>{e   E} [+   -] digit...</pre>

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 macro 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 caret 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). The following table lists the control characters and their corresponding ASCII values.

## Chapter 2 Understanding the Basics of CASL

---

ASCII Control Codes      The following table lists ASCII control codes and corresponding control values.

ASCII	Control+Character	Name	Description
0	^@	NUL	Null
1	^A	SOH	Start of header
2	^B	STX	Start of text
3	^C	ETX	End of text
4	^D	EOT	End of transmission
5	^E	ENQ	Enquiry
6	^F	ACK	Positive acknowledgment
7	^G	BEL	Bell
8	^H	BS	Backspace
9	^I	HT	Horizontal tab
10	^J	LF	Line feed
11	^K	VT	Vertical tab
12	^L	FF	Form feed
13	^M	CR	Carriage return
14	^N	SO	Shift out
15	^O	SI	Shift in
16	^P	DLE	Data link escape
17	^Q	DC1	Device control 1
18	^R	DC2	Device control 2
19	^S	DC3	Device control 3
20	^T	DC4	Device control 4
21	^U	NAK	Negative acknowledgment
22	^V	SYN	Synchronous idle
23	^W	ETB	End of transmission block
24	^X	CAN	Cancel
25	^Y	EM	End of medium
26	^Z	SUB	Substitute

ASCII	Control+Character	Name	Description
27	^[	ESC	Escape
28	^\	FS	File separator
29	^]	GS	Group separator
30	^^	RS	Record separator
31	^_	US	Unit separator

To send a control code, use the Control+Character value or the name listed in the preceding table. If you use the name, be sure to enclose it in angle brackets. For example, you can use `^[` or `<ESC>` to represent the ASCII code for Escape. The macro processor interprets this as the Escape code 1B hexadecimal.

To send the code as a string, precede it with a grave accent (```).

### Special Characters

Some characters have special meanings to Accessory Manager's CASL processor. If you want a special character to be recognized as part of a string constant, precede the character with a grave accent.

This is illustrated in the following examples:

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

In the first example, the macro processor interprets the `"|"` as a carriage return. In the second example, the macro processor interprets `"`|"` as the vertical bar character.

The special characters are as follows:

Character	Special Meaning to the CASL Processor
"	Double quotation mark. Delimiter around a string constant.
'	Single quotation mark. Delimiter around a string constant.
\	Backslash. Precedes an ASCII value.
	Vertical bar. A carriage return.
`	Grave accent. Marks special characters in a string.
^	Caret. Precedes another character to denote an ASCII control character, as in ^A for start of header control character.
<	Less-than symbol. Used to mark the beginning of a keystroke name .

If you want a grave accent to be recognized as part of the string, precede it with another grave accent. The first one protects the second.

Using the grave accent with these special characters is essential when using the `wait` statement to wait for a string that contains these characters. Refer to “`wait (statement)`” on page 344.

When working with a block mode terminal emulator, such as T 27 or UTS, you often need to use the grave accent in a `press` or `reply` statement that includes control characters. Refer to “`press (statement)`” on page 281 and “`reply (statement)`” on page 297.

#### Keystroke Names

If you need to specify a key on the PC keyboard or a terminal emulation keystroke in a string constant, enter it as follows:

```
"`<Transmit>"
```



**String Constants  
That Continue on a  
New Line**

When 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 shows 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 macro 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. Sub-expressions inside parentheses are evaluated before the main expression.

The general precedence of operators is as follows:

- Highest—Arithmetic and string operators
- Next highest—Relational operators
- Lowest—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.

## 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	Rot
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
```

`Rot`, `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.

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 `Rot` 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 a plus sign (+).

When you use the string concatenation operator, two strings connected by a plus sign (+) 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, refer to [Chapter 5, “Functional Purpose of CASL Elements,”](#) and [Chapter 6, “CASL 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:

Operator	Description
=	Equal
>=	GreaterOrEqual
>	GreaterThan
<>	Inequality
<=	LessOrEqual
<	LessThan

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 `not`. The operator with the next highest precedence is `and`. The operator with the lowest precedence is `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(f1) 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 `0` 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 four characters long; otherwise, it is eight 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 displays the result on the screen.

**Converting an  
ASCII Value to a  
Character String**

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

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

In this 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, refer to [Chapter 5, “Functional Purpose of CASL Elements,”](#) and [Chapter 6, “CASL Language.”](#)

## Compiler Directives

Compiler directives provide instructions for the macro 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 macro description

### Suppressing Label Information

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

### Suppressing Line Number Information

Information about line numbers is also included as part of a compiled macro. To suppress this information, add the `genlines off` compiler directive at the beginning of your macro. 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, refer to [Chapter 4, "Interacting with the Host, Users, and Other Macros."](#)

**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 macro being compiled. The file is included in the macro 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 Macro Description**

Use the `scriptdesc` compiler directive to define descriptive text for a macro.

For more detailed information about these compiler directives, refer to [Chapter 6, “CASL 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 statements (such as `watch`), words that define time (such as `seconds` and `ticks`), and words that bind statements (such as `for` and `next`).

The following are the CASL keywords.

<code>abs</code>	<code>accept</code>	<code>across</code>
<code>activate</code>	<code>activatesession</code>	<code>active</code>
<code>alarm</code>	<code>alert</code>	<code>align</code>
<code>alluc</code>	<code>and</code>	<code>answer</code>
<code>append</code>	<code>arg</code>	<code>arrow</code>
<code>as</code>	<code>asc</code>	<code>assume</code>
<code>at</code>	<code>attr</code>	<code>aux</code>
<code>backups</code>	<code>binary</code>	<code>bitstrap</code>
<code>bitstrip</code>	<code>black</code>	<code>blue</code>
<code>bol</code>	<code>bool</code>	<code>Boolean</code>
<code>border</code>	<code>bow</code>	<code>box</code>
<code>bright</code>	<code>brown</code>	<code>browse</code>
<code>builtin</code>	<code>busycursor</code>	<code>bye</code>
<code>byte</code>	<code>call</code>	<code>cancel</code>
<code>capacity</code>	<code>capture</code>	<code>case</code>
<code>cd</code>	<code>chain</code>	<code>char</code>
<code>char</code>	<code>chdir</code>	<code>checkbox</code>
<code>chmod</code>	<code>choice</code>	<code>choices</code>
<code>chr</code>	<code>cksum</code>	<code>class</code>
<code>clear</code>	<code>close</code>	<code>cls</code>
<code>cmode</code>	<code>color</code>	<code>compile</code>
<code>connected</code>	<code>connectreliable</code>	<code>copy</code>



## Chapter 2 Understanding the Basics of CASL

---

count	crc	ctext
curday	curdir	curdrive
curhour	curminute	curmonth
cursecond	curyear	cyan
date	default	definput
defoutput	defpushbutton	dehex
delay	delete	deletesubstring
description	destore	detext
device	devicevar	dialmodifier
dialogbox	dir	direct
diskspace	display	do
down	draw	drive
drop	echo	edit
editor	edittext	else
end	endcase	enddialog
endfunc	endproc	endwatch
enhex	enstore	entext
environ	eof	ej
eol	eop	eow
errclass	errno	error
exec	exists	exit
extern	external	fail
false	field	fileattr
filedate	filefind	filesize
filetime	fill	filter
filtervar	fkey	flashing
flood	fncheck	fnstrip
focus	footer	for
form	forward	freefile
freemem	freetrack	from

## Chapter 2 Understanding the Basics of CASL

---

func	function	genlabels
genlines	get	getnextline
global	go	gosub
goto	gray	green
group	groupbox	halt
header	height	help
hex	hidden	hide
hideallquickpads	hidequickpad	hms
hollow	hour	hours
if	include	index
inject	inkey	input
inscript	insert	instr
integer	intval	inverse
is	isnt	istrackhit
jump	keep	key
keys	label	left
leftjustify	len	length
library	lift	line
listbox	load	loadquickpad
loc	locked	lowcase
lprint	ltext	magenta
match	max	maximize
maxlength	md	message
mid	millisecond	min
minimize	minus	minute
minutes	mkdir	mkint
mkstr	mod	modem
move	name	netid
new	next	nextchar
nextline	noask	noblanks

## Chapter 2 Understanding the Basics of CASL

---

noby	nocase	none
nopause	normal	not
null	number	octal
of	off	offset
ok	on	online
only	ontime	open
optional	or	output
over	pack	pad
page	paint	pan
password	pause	perform
picture	plus	pop
preserve	press	print
printer	proc	procedure
prompt	protocol	protocolvar
public	pure	pushbutton
put	quiet	quit
quote	radiobutton	random
rd	read	real
receive	red	redialcount
redialwait	release	remove
rename	repeat	replace
reply	request	reset
restore	resume	return
returns	reverse	right
rmdir	rol	ror
routine	rtext	run
save	script	scriptdesc
scroll	secno	second
seconds	seek	send
sendbreak	session	sessionvar

## Chapter 2 Understanding the Basics of CASL

---

sessname	sessno	setup
setvar	shl	show
showquickpad	shr	shut
size	slice	some
sort	space	start
startup	statevar	static
status	step	str
string	strip	stripclass
stripwild	stroke	style
subst	subtitle	swap
systemvar	stime	tabstop
tabwidth	takes	terminal
terminalvar	terminate	then
tick	ticks	time
timeout	times	title
to	toggle	trace
track	trackhit	trap
true	type	unloadallquickpads
unloadquickpad	until	up
upcase	userid	val
version	view	viewport
wait	watch	weekday
wend	while	white
width	winchar	window
winsize	winsizey	winstring
winversion	word	write
xpos	xsep	yellow
yourself	ypos	ysep
zone	zoom	

---

# *Variables, Arrays, Procedures, and Functions*

# 3

## **In This Chapter**

In a CASL macro, you use declarations to define your variables, arrays, procedures, and functions. Declarations make your macro more readable and maintainable; in some instances, they are mandatory.

This chapter contains information about declaring elements in the CASL language. It includes the following headings:

Variables .....	64
Explicit Variable Declarations .....	65
Implicit Variable Declarations .....	67
Arrays .....	68
Procedures .....	70
Functions .....	73
Scope Rules .....	75
Calling DLL Functions .....	77

## Variables

A variable is a language element whose value can change during the course of running a macro. 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 macro)
- User-defined variables (which you define in your macro)

### Predefined Variables

There are two types of predefined variables:

- System variables
- Module variables

System variables contain user-profile (or configuration) information or session information. 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 contain tool-specific information and are stored in a session profile. For example, if a session uses the INFOConnect connection tool, the entry contains settings for INFOConnect paths and so on. To reference these variables, use the `assume` statement as follows:

```
assume device "ICSTOOL"
```

### User-Defined Variables

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

## Explicit Variable Declarations

Explicitly declare your variables to make your macro 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 macro. 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 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
```

### 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`.

### Public and External Variables

If you want to share a variable among multiple macros, declare the variable as `public` in the main macro (parent macro) and as `external` in the other macros (child macros). The data type of the variables must match. 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 macro declaration)

external integer user_name    (child macro declaration)
```

For additional information about `public` and `external` variables, refer to [Chapter 4, “Interacting with the Host, Users, and Other Macros.”](#)



## Implicit Variable 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 macro 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 following example, `user_name` is implicitly declared as a string variable because the string "John" is assigned to it. "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
```

## 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-Dimensional 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).

### Multidimensional Arrays

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 macro 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, passwd  
    reply username  
    wait 2 seconds  
    reply passwd  

```

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 `passwd`—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 macro. 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 macro. 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 shows how to set up the logon procedure using a forward declaration:

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

logon userid, password        -- The procedure call

proc logon takes username, passwd -- The procedure
    reply username
    wait 2 seconds
    reply passwd
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 macro, 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 macro you create.

To include an external procedure in a macro, 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 macro, add the following line at the beginning of the macro:

```
include "myprocs"
```

For more information about the proc...endproc procedure construct, the perform statement, and the include compiler directive, refer to [Chapter 6, "CASL 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 macro. 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 macro. 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 do. 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 shows how to set up the `calc` function using a forward declaration:

```
integer return_value      -- The integer declaration  
  
func calc(integer x, integer y) ... -- The forward  
  returns integer 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 macro, use the `include` compiler directive. For example, if the `calc` function is external to the macro and is stored in a file called `MYPROCS.XWS`, add the following line at the beginning of the macro to include it in the macro:

```
include "myprocs"
```

For more information about the `func...endfunc` function and the `include` compiler directive, refer to [Chapter 6, "CASL 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 macro ends. You can refer to and modify these variables within and outside procedures and functions. They retain their values throughout execution of the macro.

### 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:

Data Type	Default Value
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 macro 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 macro ends. Procedures and functions cannot reference labels that are not defined within the procedure or function.

## Calling DLL Functions

In a CASL macro, you can call functions located in external libraries. These libraries are referred to as Dynamic Link Libraries (DLLs) in the Windows environment. This provides access to Windows' kernel, user, or GDI functions, third-party libraries, and in-house libraries. The advantage of using external libraries is to provide capabilities not found in CASL and to improve the efficiency of critical routines.

An external library is a collection of functions that exist in a separate file. That file is loaded by the operating system only when a program (or macro in our case) calls one of the functions contained in it. This reduces the size of programs, since many programs can call the same library, and allows new functionality to be added to CASL.

**Note:** The following information is intended for experienced Windows programmers.

### Declaring DLL Functions

The functions in your CASL macro that call DLL functions are declared in a manner similar to CASL forward declarations. Once declared, the functions can be used exactly like other functions in your macro. Use the following syntax to declare the functions:

Function with a return value:

```
func name [(arglist)] returns type ...  
  library filename [name (string)]
```

Procedure without a return value:

```
proc name [takes arglist ] library filename ...  
  [name (string)]
```

The name can be the real name of the function or a name preferred by the user. In the latter case, the optional name parameter at the end of the declaration must provide the real function name.

The following examples illustrate library declarations:

```
func IsCharAlpha(char x) returns boolean ...  
  library "user.exe"
```

USER.EXE is one of the DLLs that comprise the Windows core.

```
func myFunc(integer x, real y) returns integer ...
    library "mylib.dll" name "FredFunc"

func countLetters(string x) returns integer ...
    library "stringlib.dll"

proc do_something takes integer x, byte y,
    string z ...
    library "something.dll"
```

**Note:** Since the functions are only declared in the macro, the parameter names used in the declarations (*x*, *y*, and *z*) are place holders and can be any valid variable name. Make sure you include the file name extension .DLL. Also, a path is required if the DLL is not located in any directory that is searched automatically by Windows.

### Parameter and Return Values

The following CASL data types can be passed as parameters to DLL functions: integer, real, string, boolean, byte, and word.

The list is the same for return values with the exception of real, which is not returned.

A DLL function is written in a language such as C/C++. You need to match the CASL data type to the data type expected by the function being called.

Use the following table to select the data type you need.

<b>C or C++ Data Type</b>	<b>CASL's Corresponding Type</b>
long (32 bit data)	integer
unsigned long	integer
int (16 bit data)	word
short	word
unsigned int or short	word
char (numeric value)	\word
char (single letter)	char
unsigned char	byte
float	not supported
double	real
char * (pointer to char)	string

CASL integers are 32-bit signed values, CASL words and bytes are 16- and 8-bit unsigned values respectively. Keep this in mind when assigning values to variables. Where a function takes or returns an 8- or 16-bit value that is designated as true or false, you can define it as Boolean and use the true or false keywords built into CASL.

**Note:** CASL does not pass the string itself to a function. Instead the location (address) of the string is passed. In C, this is referred to as a pointer. In CASL, you simply use a string variable as a parameter or place the desired text in quotes. CASL handles the job of passing the correct information.

Sometimes functions use the pointer as a method of returning data over and above the return value. Since the function has the location of the string, it can write data to that location. For example, a function that converts text to uppercase might simply do the job “in place,” so that the string you passed as a parameter is also the string that contains the uppercase text. In this case, a string variable must be used so that you can reference the string later.

You need to make sure that data returned in the string does not exceed the length of the original string. For example, you may have a function named `path` that takes the name of a file as a parameter and returns, in the same string, the full path specification for that file, as follows:

```
string file_str
file_str = "myfile.txt"
path (file_str)
```

In this case, you will get a truncated path name if it is longer than the string. The function assumes it has enough space and will write beyond the end of your original string. This can corrupt your data or lock up your computer. The following macro shows the correct approach, making the string long enough to accommodate the longest string anticipated (in the case of DOS path names, 128 characters).

```
string file_str
file_str = "myfile.txt"      -- Add extra blanks
path (file_str)
strip (file_str, " ", 1)    -- Remove excess blanks
```

### **Non-Supported Parameters and Return Values**

Functions written in languages such as C can accept a wide range of parameters not supported by CASL DLL calls, such as arrays and structures. If you want to access such functions (for example, in third-party libraries), you must write intermediate libraries that translate the data being passed or returned.

### **Writing Windows DLLs**

Before you write DLLs, you should have experience with a language such as C and have access to a compiler that supports Windows programming.

To access functions in a DLL, a DEF file must export each of the callable functions in its EXPORTS section. If you are not already familiar with writing DLLs, you should refer to the books available that provide detailed explanations of how to program Windows applications and DLLs.

**Note:** For string handling, remember that you are only returning the address of the string. This means that the address must remain valid after the function ends. Do not return a local string (one on the stack). Declare any string to be returned as static or allocate it from heap memory. However, if any memory is allocated on the heap, whether for strings or for any other data, it must be freed at some point before the macro terminates. Therefore, you must free the memory from the function that allocates it or provide another function to free it.

As you write DLLs to interface to Windows, you might need access to Accessory Manager parent and child (session) window handles. To access these handles, declare the following at the top of the macro:

```
/*Handle to Accessory Manager parent window */
systemvar integer _hWndFrame

/* Handle to script's child window */
sessionvar integer _hWndSession
```

After the declaration, `_hWndSession` and `_hWndFrame` are used in the same manner as `system` and `session` variables.





---

# *Interacting with the Host, Users, and Other Macros*

# 4

## **In This Chapter**

This chapter includes the following headings:

Interacting with the Host .....	84
Communicating with a User .....	87
Invoking Other Macros .....	90
Exchanging Variables .....	91
Trapping and Handling Errors .....	92

## Interacting with the Host

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.

### Waiting for a Character String

Use the `wait` statement when you need to wait for a specific, unique string of text, as in the following 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 macro 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 macro 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 macro 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 macro waits two seconds for the host to send the `login:` prompt. If a timeout occurs before the prompt appears, the user is alerted and the macro ends.

By default, the `wait` statement is not case- or space-sensitive. If your macro 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. For a complete list of `wait` conditions, refer to “[wait \(statement\)](#)” on page 344.

### Watching for 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, as shown in the following example:

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

In this example, when the `watch` statement is encountered, the macro pauses while waiting for one of the two 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 macro 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 macro ends if the user presses `Esc` (key 27). If `more:` is found, the macro waits one 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 macro ends.

Like the `wait` statement, the `watch` statement can watch for several different kinds of conditions. For a complete list of the conditions, refer to “`watch...endwatch (statements)`” on page 348.

### 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, macro execution continues with the statement following the `repeat...until` construct.

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

### Sending a Reply to the Host

Many of the examples in this section use the `reply` statement to respond to the host. 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, your macros 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`.

### Displaying Information

Use the `print` statement to display information in the session window. You can display constants, variables, or a combination of the two. You can also control such display characteristics as attributes for bright or flashing characters and for color. Note that attributes will work only if the terminal type, which controls the interface between the macro 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, as 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. 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 PC 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. For a list of other ASCII control characters, refer to “[ASCII Control Codes](#)” on page 40.

## Requesting Information

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

The following example shows 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 let you create dialog boxes for text input. The `alert` statement displays a simple dialog box in which the user can enter text or respond by clicking a button. The `dialogbox...enddialog` construct lets you create more sophisticated dialog boxes, which can contain buttons, 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` appears in the dialog box. The user can click either `Try Again`, `Cancel`, or `OK` to exit the dialog box.

For additional information about the `print`, `message`, `input`, `alert`, and `dialogbox...enddialog` statements, refer to [Chapter](#), “[CASL Language](#).”

## Invoking Other Macros

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

### Chaining to Another Macro

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

```
chain "SCRIPT2"
```

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

### Calling Another Macro

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

```
do "cvtsrc"
```

### Passing Arguments

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

```
chain "SCRIPT2 CSERVE"
```

To retrieve the arguments in the invoked macro, use the `arg` function. Use `arg` with no arguments (or an argument of 0) 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 macro, the macros can exchange variable information. To pass a variable between macros, declare the variable as `public` in the invoking macro and as `external` in the invoked macro.

In the following example, the invoking macro, `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 macro if you use the `chain` statement to invoke the macro. Also, if you are using `public` and `external` variables, you must declare the variable as `public` in the parent macro.

## Trapping and Handling Errors

Error trapping makes a macro capable of handling almost any situation, and it is essential in macros 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.

### Enabling Error Trapping

Use the `trap` compiler directive to enable and disable error trapping in your macro. The default setting for this directive is `trap off`. If `trap` is `off`, a dialog box appears automatically and the macro ends whenever a fatal error occurs. If `trap` is `on`, the dialog box does not appear, and the macro continues running.

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`, 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.

### 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 0. To continue to trap errors throughout the execution of the macro, 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 0 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 [“Classes of Error Message”](#) on page 380.

### 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` system 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, refer to [Appendix A, “Error Messages.”](#)

If no error occurs, the `errno` variable contains 0. This variable is not reset when you check its value; the value remains unchanged until a different error occurs.

When setting up your macro 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 macro 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 macro ends. If for any reason the file transfer does not complete successfully, the macro sounds an alarm and prints an error message.

```
/* Macro 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 macro is very simple. Ideally, your error handling should be more comprehensive. For example, if the macro 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 macro 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 macro needs to accomplish.

Refer to [Chapter 6, “CASL Language,”](#) for more information on the `trap` compiler directive, the `error` function, and the `errclass` and `errno` system variables.

---

# ***Functional Purpose of CASL Elements***

# **5**

## **In This Chapter**

This chapter groups CASL macro elements by function and includes the following headings:

Overview .....	96
Date and Time Operations .....	97
Error Control .....	98
File Input/Output Operations .....	99
Host Interaction .....	101
Macro Management .....	102
Mathematical Operations .....	103
Printer Control .....	104
Program Flow Control .....	105
Session Management .....	107
String Operations .....	109
Type Conversion Operations .....	111
Window Control .....	112
Miscellaneous Elements .....	114

## Overview

This chapter contains a quick reference to all of the CASL elements. Detailed descriptions of the elements and examples showing how to use them are covered in [Chapter , “CASL Language.”](#)

In this chapter, CASL elements are grouped according to their functional purpose, such as session management, program flow control, file input/output operations, and so on. Some elements might 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:

Identifier	Macro Element Group
F	Function
S	Statement
V	Variable (system and module)
C	Constant
D	Declaration (procedure and function)
CD	Compiler directive

## Date and Time Operations

The following CASL elements determine the date and time:

<b>Element</b>	<b>Description</b>
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)

## Error Control

The following CASL elements control error conditions:

<b>Element</b>	<b>Description</b>
<code>errclass</code>	Indicates the class of the last error. (V)
<code>errno</code>	Indicates the type of the last error. (V)
<code>error</code>	Indicates the occurrence of an error. (F)
<code>trap</code>	Turns error trapping on and off. (CD)



## File Input/Output Operations

The following CASL elements provide file input and output capabilities:

Element	Description
backups	Determines what is done with duplicate files after a file transfer. (V)
chdir	Changes to a different disk directory. (S)
close	Closes a disk file. (S)
copy	Copies a file or group of files. (S)
curdir	Returns the current disk directory. (F)
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)
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)
filefind	Locates files in the directory. (F)
filesize	Returns the file size. (F)
fncheck	Checks the validity of a file name. (F)
fnstrip	Returns specified portions of a file name. (F)
get	Reads characters from a random access file. (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)

**Chapter 5** *Functional Purpose of CASL Elements*

---

<b>Element</b>	<b>Description</b>
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)
write	Writes text fields to a file. (S)
write line	Writes text lines to a file. (S)

## Host Interaction

The following CASL elements let you interact with a host:

Element	Description
<code>display</code>	Controls the display of incoming characters. (V)
<code>match</code>	Specifies the string found by the last <code>wait</code> or <code>watch</code> statement. (V)
<code>nextchar</code>	Returns the next character from a communications device. (F)
<code>nextline</code>	Returns the next line, delimited by a carriage return, from the communications device. (F/S)
<code>online</code>	Returns true if a session is online. (F)
<code>press</code>	Sends a series of keystrokes to the terminal module. (S)
<code>reply</code>	Sends a string of text to the communications device. (S)
<code>sendbreak</code>	Sets the length of a break signal. (S)
<code>track</code>	Watches for string patterns or keystrokes while online. (S)
<code>wait</code>	Waits for a string of text from the communications device or for a keystroke. (S)
<code>watch...endwatch</code>	Watches for one of several conditions to occur. (S)

## Macro Management

The following CASL elements manage CASL macros:

<b>Element</b>	<b>Description</b>
chain	Passes control to another macro. (S)
compile	Compiles a macro. (S)
do	Starts another macro and waits for it to return control. (S)
genlabels	Specifies whether to include or exclude label information in a compiled macro. (CD)
genlines	Specifies whether to include or exclude line information in a compiled macro. (CD)
include	Includes an external file in a compiled macro. (CD)
inscript	Checks for labels in a macro. (F)
quit	Closes a session window. (S)
scriptdesc	Defines a macro description. (CD)
startup	Contains the name of the macro to run at start-up. (V)
terminate	Terminates Accessory Manager. (S)
trace	Turns tracing on and off. (S)

## Mathematical Operations

The following CASL elements perform mathematical operations:

<b>Element</b>	<b>Description</b>
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 following CASL elements control how data is printed:

<b>Element</b>	<b>Description</b>
capture	Sends a continuous stream of data from the host to a file. (S)
footer	Specifies the footer used when printing. (V)
grab	Sends the contents of the session window to a file. (S)
header	Specifies the header used when printing. (V)
lprint	Sends a string of text to the printer. (S)
printer	Sends a continuous stream of data from the host to a printer. (V)

If you use a CASL macro that contains the `PRINTER=ON`, `PRINTER=OFF`, or `CAPTURE NEW` commands, the Capture Mode must be Raw. To set this option, click Capture from the File menu, click Options, click Raw, click OK, and then click OK again.

## Program Flow Control

The following CASL elements provide program flow control:

Element	Description
<code>case...endcase</code>	Performs statements based on the value of a specified expression. (S)
<code>chain</code>	Passes control to another macro. (S)
<code>do</code>	Starts another macro and waits until it returns control. (S)
<code>end</code>	Ends a macro. (S)
<code>exit</code>	Exits a procedure. (S)
<code>for...next</code>	Performs a series of statements a specified number of times, usually while changing the value of a variable. (S)
<code>freetrack</code>	Returns the value of the lowest unused track number for the current session. (F)
<code>func...endfunc</code>	A function declaration. (D)
<code>gosub...return</code>	Transfers program control to a subroutine. (S)
<code>goto</code>	Transfers program control to a label or expression. (S)
<code>halt</code>	Stops a macro and its related parent and child macros. (S)
<code>if...then...else</code>	Controls program flow based on the value of an expression. (S)
<code>label</code>	Denotes a named reference point in a macro. (S)
<code>perform</code>	Calls a procedure. (S)
<code>proc...endproc</code>	A procedure declaration. (D)
<code>quit</code>	Closes a session window. (S)
<code>repeat...until</code>	Repeats a statement or series of statements until a specified condition is true. (S)
<code>return</code>	Returns a value from a function. (S)
<code>terminate</code>	Terminates Accessory Manager. (S)
<code>timeout</code>	Returns the status of the most recent wait or watch statement. (V)

**Chapter 5** *Functional Purpose of CASL Elements*

---

<b>Element</b>	<b>Description</b>
<code>trace</code>	Turns tracing on and off. (S)
<code>track</code>	Watches for string patterns or keystrokes while online. (S)
<code>wait</code>	Waits for a string of text from the communications device or for a keystroke. (S)
<code>watch...endwatch</code>	Watches for one of several conditions to occur. (S)
<code>while...wend</code>	Performs a statement or group of statements as long as a specified condition is true. (S)



## Session Management

The following CASL elements manage sessions:

Element	Description
<code>activate</code>	Activates Accessory Manager by moving the focus to it. (S)
<code>activatesession</code>	Makes the specified session active. (S)
<code>assume</code>	Controls the way the CASL compiler handles module variables for the Connection, Terminal, and File Transfer tools. (S)
<code>bye</code>	Disconnects the current session. (S)
<code>description</code>	Describes a session. (V)
<code>device</code>	Specifies a connection device. (V)
<code>go</code>	Initiates a connection to a communications device. (S)
<code>keys</code>	Reads or sets the keyboard map to use. (V)
<code>name</code>	Contains the name of the current session. (F)
<code>netid</code>	Contains the network identifier for a session. (V)
<code>new</code>	Creates or opens a session. (S)
<code>ontime</code>	Indicates how long a session has been online. (F)
<code>password</code>	Contains the password for the current session. (V)
<code>protocol</code>	Specifies a file transfer protocol. (V)
<code>quit</code>	Closes a session window. (S)
<code>run</code>	Starts another application. (S)
<code>save</code>	Saves the current session parameters. (S)
<code>script</code>	Specifies the name of the macro file to use for the current session. (V)
<code>session</code>	Returns the session number of the current session. (F)
<code>sessname</code>	Returns the name of the session identified by a specified session number. (F)
<code>sessno</code>	Returns the session number of a specified session. (F)

**Chapter 5** *Functional Purpose of CASL Elements*

---

<b>Element</b>	<b>Description</b>
startup	Contains the name of the macro to run at start-up. (V)
terminal	Specifies the terminal emulation to use. (V)
terminate	Terminates Accessory Manager. (S)
userid	Contains the user account name for a session. (V)

## String Operations

The following CASL elements perform string operations:

Element	Description
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)
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)
lowercase	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)

**Chapter 5** *Functional Purpose of CASL Elements*

---

<b>Element</b>	<b>Description</b>
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 CASL elements convert data from one type to another:

Element	Description
<code>asc</code>	Returns the ASCII value of a string. (F)
<code>binary</code>	Converts a string to a binary number. (F)
<code>bitstrip</code>	Strips bits from strings. (F)
<code>chr</code>	Returns a single-character string for an ASCII value. (F)
<code>class</code>	Returns the class type of a single-character string. (F)
<code>dehex</code>	Converts ASCII strings in hexadecimal format to binary. (F)
<code>detext</code>	Converts 7-bit ASCII character strings to binary. (F)
<code>enhex</code>	Converts a binary string to a string of ASCII characters in hexadecimal format. (F)
<code>entext</code>	Converts a string of binary data to a string of 7-bit ASCII characters. (F)
<code>hex</code>	Converts an integer to a hexadecimal string. (F)
<code>intval</code>	Returns the integer value of a string. (F)
<code>mkint</code>	Converts numeric strings to integers. (F)
<code>mkstr</code>	Converts an integer to a string. (F)
<code>octal</code>	Converts a decimal integer to an octal integer. (F)
<code>str</code>	Converts a number to string format. (F)
<code>val</code>	Returns the real (floating point) value of a string. (F)

## Window Control

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

Element	Description
activate	Activates Accessory Manager window by moving the focus to it. (S)
alert	Creates simple dialog boxes for display on the screen. (S)
choice	Contains the value of the button 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 Accessory Manager window to full-screen size. (S)
minimize	Reduces the Accessory Manager window to an icon. (S)
move	Moves the Accessory Manager window to a new location on the screen. (S)
passchar	Specifies the character to display in a text box on a dialog box created using dialogbox...enddialog and the secret option. (V)
print	Displays information on the screen. (S)
restore	Restores the Accessory Manager window to its original size. (S)
show	Redisplays a session window. (S)
showquickpad	Displays a QuickPad. (S)

---

**Chapter 5** *Functional Purpose of CASL Elements*

---

<b>Element</b>	<b>Description</b>
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 Accessory Manager application window. (S)

## Miscellaneous Elements

The following are CASL elements that don't fall into the preceding categories:

<b>Element</b>	<b>Description</b>
alarm	Sounds an alarm at the terminal. (S)
busycursor	Displays the cursor as an hourglass. (S)
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)
system	Indicates how long the current session has been active. (F)
true	Sets a variable to logical true. (C)
version	Returns the Accessory Manager version number. (F)
winversion	Returns the Windows version number. (F)



---

# CASL Language

# 6

## In This Chapter

This chapter provides detailed information about all CASL elements, including the syntax of each element and examples of how the element can be used.

How CASL Elements Are Documented . . . . .	124
abs (function) . . . . .	125
activate (statement) . . . . .	126
activatesession (statement) . . . . .	127
alarm (statement) . . . . .	128
alert (statement) . . . . .	130
arg (function) . . . . .	132
asc (function) . . . . .	133
assume (statement) . . . . .	134
backups (module variable) . . . . .	135
bitstrip (function) . . . . .	137
busycursor (statement) . . . . .	138
bye (statement) . . . . .	139
capture (statement) . . . . .	140

chain (statement) .....	144
chdir (statement).....	145
choice (system variable) .....	146
chr (function).....	147
cksum (function) .....	148
class (function) .....	149
clear (statement).....	150
close (statement).....	151
cls (statement).....	152
compile (statement) .....	153
connected (function) .....	154
copy (statement) .....	155
count (function).....	156
crc (function) .....	157
curday (function).....	158
curdir (function) .....	159
curdrive (function) .....	160
curhour (function).....	161
curminute (function).....	162
curmonth (function) .....	163
cursecond (function) .....	164
curyear (function) .....	165
date (function).....	166
definput (system variable) .....	167
defoutput (system variable) .....	168
dehex (function).....	169
delete (statement).....	170

delete (function) . . . . .	171
description (system variable). . . . .	172
destore (function) . . . . .	173
detext (function) . . . . .	174
device (system variable). . . . .	175
dialogbox...enddialog (statements) . . . . .	176
display (system variable). . . . .	183
do (statement). . . . .	184
drive (statement) . . . . .	186
end (statement). . . . .	187
enhex (function) . . . . .	188
enstore (function) . . . . .	189
entext (function) . . . . .	190
environ (function) . . . . .	191
eof (function). . . . .	192
eol (function). . . . .	193
errclass (system variable) . . . . .	195
errno (system variable) . . . . .	196
error (function) . . . . .	197
exists (function) . . . . .	198
exit (statement) . . . . .	199
false (constant) . . . . .	200
filefind (function) . . . . .	201
filesize (function) . . . . .	203
fncheck (function) . . . . .	204
fnstrip (function). . . . .	205
footer (system variable) . . . . .	207

for...next (statements).....	208
freemem (function) .....	210
freetrack (function) .....	211
func...endfunc (function declaration) .....	212
genlabels (compiler directive).....	214
genlines (compiler directive) .....	215
get (statement) .....	216
go (statement) .....	217
gosub...return (statements) .....	218
goto (statement) .....	219
grab (statement) .....	220
halt (statement) .....	221
header (system variable) .....	222
hex (function) .....	223
hide (statement) .....	224
hideallquickpads (statement).....	225
hidequickpad (statement).....	226
hms (function).....	227
homedir (system variable) .....	228
if...then...else (statements).....	229
include (compiler directive) .....	231
inject (function).....	232
inkey (function).....	233
input (statement) .....	235
inscript (function) .....	236
insert (function).....	237
instr (function) .....	238

intval (function) . . . . .	239
jump (statement) . . . . .	240
keys (system variable) . . . . .	241
label (statement) . . . . .	242
left (function) . . . . .	243
length (function) . . . . .	244
loadquickpad (statement) . . . . .	245
loc (function) . . . . .	246
lowercase (function) . . . . .	247
lprint (statement) . . . . .	248
match (system variable) . . . . .	249
max (function) . . . . .	250
maximize (statement) . . . . .	251
mid (function) . . . . .	252
min (function) . . . . .	253
minimize (statement) . . . . .	254
mkdir (statement) . . . . .	255
mkint (function) . . . . .	256
mkstr (function) . . . . .	257
move (statement) . . . . .	258
name (function) . . . . .	259
netid (system variable) . . . . .	260
new (statement) . . . . .	261
nextchar (function) . . . . .	262
nextline (statement) . . . . .	263
nextline (function) . . . . .	265
null (function) . . . . .	267

octal (function) .....	268
off (constant) .....	269
on (constant) .....	270
online (function) .....	271
ontime (function).....	272
open (statement).....	273
pack (function).....	274
pad (function) .....	275
passchar (system variable).....	277
password (system variable) .....	278
perform (statement) .....	279
pop (statement).....	280
press (statement) .....	281
print (statement).....	283
printer (system variable) .....	284
proc...endproc (procedure declaration).....	285
protocol (system variable).....	288
put (statement) .....	289
quit (statement) .....	290
quote (function).....	291
read (statement) .....	292
read line (statement) .....	293
receive (statement) .....	294
rename (statement).....	295
repeat...until (statements) .....	296
reply (statement).....	297
request (statement).....	298

restore (statement) . . . . .	299
return (statement) . . . . .	300
right (function) . . . . .	301
rmdir (statement) . . . . .	302
run (statement) . . . . .	303
save (statement) . . . . .	304
script (system variable) . . . . .	305
scriptdesc (compiler directive) . . . . .	306
secno (function) . . . . .	307
seek (statement) . . . . .	308
send (statement) . . . . .	309
sendbreak (statement) . . . . .	310
session (function) . . . . .	311
sessname (function) . . . . .	312
sessno (function) . . . . .	313
show (statement) . . . . .	314
showallquickpads (statement) . . . . .	315
showquickpad (statement) . . . . .	316
size (statement) . . . . .	317
slice (function) . . . . .	318
startup (system variable) . . . . .	319
str (function) . . . . .	320
strip (function) . . . . .	321
stroke (function) . . . . .	322
subst (function) . . . . .	323
systemtime (function) . . . . .	324
tabwidth (module variable) . . . . .	325

terminal (system variable) . . . . .	326
terminate (statement) . . . . .	327
time (function) . . . . .	328
timeout (system variable) . . . . .	329
trace (statement) . . . . .	330
track (statement) . . . . .	331
track (function) . . . . .	334
trap (compiler directive) . . . . .	336
true (constant) . . . . .	337
unloadallquickpads (statement) . . . . .	338
unloadquickpad (statement) . . . . .	339
uppercase (function) . . . . .	340
userid (system variable) . . . . .	341
val (function) . . . . .	342
version (function) . . . . .	343
wait (statement) . . . . .	344
watch...endwatch (statements) . . . . .	348
weekday (function) . . . . .	351
while...wend (statements) . . . . .	352
winchar (function) . . . . .	353
winsizeX (function) . . . . .	354
winsizeY (function) . . . . .	355
winstring (function) . . . . .	356
winversion (function) . . . . .	357
write (statement) . . . . .	358
write line (statement) . . . . .	359
xpos (function) . . . . .	360



ypos (function) ..... 361  
zoom (statement) ..... 362

## How CASL Elements Are Documented

In this chapter, all CASL elements are listed in alphabetical order. (For a summary of CASL elements grouped by function, refer to [Chapter 5, “Functional Purpose of CASL Elements.”](#))

The name of each CASL element appears as a heading at the top of the page. The type of element it is (such as function, statement, system variable, and so on) appears in parentheses.

Immediately below the CASL element name is a brief description of the element and how it should be used, followed by these sections:

- Format—the syntax for the element

**Note:** For a description of the notation used in the format, refer to [Chapter 2, “Understanding the Basics of CASL.”](#)

- Comments—additional descriptive information about the element
- Example—an example of how the element can be used
- See Also—a list of other related elements

## abs (function)

Use `abs` to get the absolute value of a number.

**Format** `x = abs(expression)`

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

**Example 1** `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`.

**Example 2** `if abs(net_worth) > 5 then alarm`

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

**See Also** `cksum`, `crc`, `intval`, `max`, `min`, `mkint`, `val`

## **activate (statement)**

Use `activate` to make the Accessory Manager application window the active window.

**Format**                    `activate`

**Example**                    `activate`

## activatesession (statement)

Use `activatesession` to make the specified session active.

**Format** `activatesession sessionid`

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

**Example 1** `activatesession sessA`

In this example, session A becomes active.

**Example 2** `activatesession sessno("ABBS")`

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

**See Also** `activate`

## alarm (statement)

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

**Format**                    `alarm [integer]`

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

*integer* can be any integer between 0 and 5; values outside of this range are treated as 0. Zero is the default value used when no argument is specified.

The sounds produced by *integer* vary, depending on the .WAV files specified in the Windows Registry in `HKEY_CURRENT_USER\AppDataEvents\Schemes\Apps\.Default`. The following table shows possible *integer* values and their corresponding sounds or registry keys:

Integer Value	Sound or Registry Key
0	Short beep
1	SystemAsterisk\Current
2	SystemExclamation\Current
3	SystemHand\Current
4	SystemQuestion\Current
5	.Default\Current

If the PC has no sound card, all the alarm values result in a beep through the speaker.

**Example 1**                    `alarm 1`

In this example, the PC makes the chord sound.

**Example 2**

```
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 macro sounds an alarm.

**Example 3**

```
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 one 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]
```

### Comments

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 buttons are displayed from left to right along the bottom of the dialog box. For *button1* through *button4*, you can use either the text that you want to display on the button or the predefined keywords `ok` and `cancel`.

If you use text for the buttons, enclose the text in quotation marks. The maximum length of a button name is ten characters.

If you use the predefined keywords `ok` and `cancel`, you do not need to enclose these keywords in quotation marks. If you use the `ok` keyword, `alert` creates an OK button in the dialog box and associates `Enter` with this button. If you use the `cancel` keyword, `alert` creates a Cancel button in the dialog box and associates `Esc` with this button.

*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 buttons 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 one and four that corresponds to the button 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 button is used to exit the dialog box.

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



**Example 1**

```
string username

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

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

**Example 2**

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

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

**See Also**

```
dialogbox...enddialog
```

## arg (function)

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

**Format** `x$ = arg[(integer)]`

**Comments** `arg` with no arguments (or an argument of 0) returns all of the arguments that follow the name of a macro in the `chain` or `do` statement. For session start-up macros, it can also return everything that was typed in the Arguments text box on the CASL Macro tab on the Session Preferences dialog box, which is accessed from Accessory Manager's Options menu.

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

**Example 1**

```
script1.xws
  do "script2", "barkley"

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

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

**Example 2**

```
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 macro file `LOGIN`. `LOGIN` reads its arguments and sends them to the host with the `reply` statement.

**See Also** `chain`, `do`

## asc (function)

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

**Format** `x = asc(string)`

**Comments** `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 -1.

**Example 1** `sixty_five = asc("A")`

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

**Example 2** `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`.

**Example 3** `x = asc(mid(thestring, 2, 1))`

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

**See Also** `binary`

## assume (statement)

Use `assume` to specify which connection, terminal, or file transfer tool is being used by the session.

**Note:** EXTRA! Enterprise for Accessory Manager sessions do not support this statement.

### Format

```
assume tool "filename" ... [, module "filename"]
```

### Comments

Before you can specify any configuration settings for a connection, terminal, or file transfer tool, you must use the `assume` statement to indicate which tool is loaded.

`tool` can be either `device` (for the connection tool), `terminal` (for the terminal tool) or `protocol` (for the file transfer tool).

`filename` is the name of the tool (it must be enclosed in quotation marks). For the connection tool, the only valid file name is `ICSTOOL`. For valid terminal tool names, refer to “[terminal \(system variable\)](#)” on page 318. For valid file transfer tool names, refer to “[protocol \(system variable\)](#)” on page 280.

For more information about connection, terminal, and file transfer tools, as well as a list of the configuration settings that you can specify for each tool, refer to [Chapter 7, “Connection, Terminal, and File Transfer Tools.”](#)

### Example

```
assume terminal "DCAT27"  
CurShape = "Block"
```

In this example, the macro indicates that the session is a T 27 session and configures the cursor shape to a block.

### See Also

`device`, `protocol`, `terminal`

## backups (module variable)

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

**Note:** Some file transfer protocols do not use this variable.

### Format

`backups = option`

### Comments

`option` is one of the following:

Option	Description
on	If 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.
off	If 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)`

**Comments** 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 the following table.

<b>Integer Value</b>	<b>Binary String Length</b>
0–255	8
256–65,535	16
65,536–2,147,483,64	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])`

**Comments** `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 (0x7F), thus stripping the high order bit from each byte in `string`. Some word processors, such as Microsoft Word, 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 0x5F (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.

**Example 1** `readable_string = bitstrip(Word_line)`

In this example, `bitstrip` strips the high-order bit of each byte of the string `Word_line` and returns the result in `readable_string`.

**Example 2** `reply bitstrip(Word_line)`

In this example, `bitstrip` strips the high-order bit of each byte of the string `Word_line`, and the result is sent to the host with the `reply` statement.

**Example 3** `all_upcase = bitstrip("abc", 0x5F)`

In this example, the letters `abc` are converted to `ABC`.

**See Also** `lowcase`, `upcase`

## busycursor (statement)

Use `busycursor` to display the cursor as an hourglass when you expect a command to take a noticeable time interval to execute.

**Format** `busycursor [on | off]`

**Comments** This statement displays the cursor as an hourglass.

**Example** `busycursor on`

```
wait 1 minute for "Login", "ID", "Password"
case match of
  "Login": reply logon
  "ID": reply userid
  "Password": reply password
endcase
```

`busycursor off`

In this example, the cursor appears as an hourglass while the match function proceeds.



## bye (statement)

Use `bye` to end a connection.

**Note:** EXTRA! Enterprise for Accessory Manager sessions do not support this statement.

**Format**

`bye`

**Comments**

This statement immediately disconnects the current session.

**Example**

```
wait for "Logged off" : bye
```

In this example, the macro waits for the phrase "Logged off" and then disconnects the session.

**See Also**

`quit`

## capture (statement)

Use capture to send screen output to a file.

**Format** `capture option [filename]`

**Comments** In Accessory Manager, clicking Capture from the File menu initiates a continuous capture of data received from the host. The capture statement performs a similar function, controlling whether data is being captured at any particular time.

When you click Capture from the File menu in Accessory Manager, you can specify whether to send the data to a printer or file. In CASL, the destination is determined by the command. Use `capture` to send a continuous stream of data to a file; use `printer` to send the data to a printer.

For the capture statement, the macro syntax determines the file name and whether the data in any existing file is overwritten or appended. However, all other settings that affect the operation of the capture (such as whether normal or raw data is captured) are controlled by the options specified on the Capture Options and Advanced Capture Options dialog boxes within Accessory Manager. (To view these dialog boxes, make sure that Show Capture Dialog When Start Capture is selected on the Global Preferences dialog box. Then click Capture from the File menu, and click Options on the Capture Printer Settings dialog box.)

Unless you specify a different drive and directory when you specify the *filename*, all files are created in the ACCMGR32 folder within the INFOConnect folder.

*option* can be any of the following values;

Option	Description
new	Starts capture and overwrites the specified file If you use the <code>new</code> option, you must specify a <i>filename</i> .
to	Starts capture and appends to the specified file If you use the <code>to</code> option, you must specify a <i>filename</i> .

Option	Description
on	<p>Starts capture with automatic file naming</p> <p>The file name is the first five letters of the session name, followed by a letter for the month (January is A, February is B, and so forth), and the day of the month. The file extension is .TXT. For example, if the session name is TCPA_1 and the file is created on April 15, the file name is TCPA_D15.TXT.</p> <p>If the file already exists, it is overwritten.</p> <p>If you use the <code>on</code> option, do not specify a <i>filename</i>. If you specify a <i>filename</i>, a compilation error will occur.</p>
pause	<p>Suspends data capture</p> <p>Data already captured is retained in a buffer. You can restart capture with the <code>capture on</code> or <code>capture toggle</code> commands, or end it with the <code>capture off</code> command.</p> <p>If you use the <code>pause</code> option, do not specify a <i>filename</i>. If you specify a <i>filename</i>, a compilation error will occur.</p>
toggle or /	<p>Toggles the capture state</p> <p>If capture is on, <code>capture toggle</code> pauses the capture. If capture is paused or off, <code>capture toggle</code> starts capture.</p> <p>If you use the <code>toggle</code> or <code>/</code> option, do not specify a <i>filename</i>. If you specify a <i>filename</i>, a compilation error will occur.</p>
off	<p>Stops data capture and closes the file</p> <p>If you use the <code>off</code> option, do not specify a <i>filename</i>. If you specify a <i>filename</i>, a compilation error will occur.</p>

**Example 1**

```
capture on
```

In this example, data is captured to a file that is automatically named based on the session name and current date.

**Example 2**

```
capture new "vutext.txt"
```

In this example, data is captured in a file called VUTEXT.TXT. If a file of that name already exists, its content is overwritten.

**See Also**

```
grab, printer
```

## 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
```

### Comments

`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 macro continues to execute at the point after the `endcase` statement (unless 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.

**Example 1**

```
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 or 4, the macro ends. If `choice` has a value between 1 and 3, the macro branches to the appropriate label. If `choice` is not 0 through 4, the default action is taken. If none of the conditions were met (assuming a default was not provided), the macro would continue execution at the statement following the `endcase`.

**Example 2**

```
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.

**See Also**

`gosub`, `goto`, `if...then...else`, `watch...endwatch`

## chain (statement)

Use `chain` to compile and run a macro.

**Format** `chain filename [, args]`

**Comments** `chain` compiles and runs the specified macro source file (.XWS) if there is no compiled version of the macro, 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 macro.

You do not have to include the macro extension, but you must include the drive and directory where the macro is located.

`args` represents an optional argument list that contains the individual arguments to be passed to the other macro. Individual arguments must be separated by commas.

**Note:** The macro that issues a `chain` statement ends and is removed from memory; therefore, control cannot be passed back to it.

**Example**

```
chain "C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32\
MENU", "arg1", "arg2"
```

In this example, the macro chains to a macro called `MENU` and passes the macro two arguments.

**See Also** `arg`, `do`

## chdir (statement)

Use `chdir` to change the current disk directory.

**Format** `chdir string`

**Comments** *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.

| **Note:** You can also use the abbreviation `cd` for this statement.

**Example 1** `chdir "C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32"`

In this example, the directory is changed to C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32.

**Example 2** `chdir dirname`

In this example, the directory is changed to the directory name stored in the script's `dirname` variable.

**See Also** `drive`

## choice (system variable)

Use `choice` to check the value of the button that dismissed a dialog box.

**Format** `n = choice`

**Comments** `choice` contains the value identifying the button used to exit a dialog box.

**Example 1**

```
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", focus
enddialog
print "Choice was "; choice
```

In this example, `choice` has a value of 1 if the Choice 1 (ok) button is chosen, 2 if the Choice 2 (cancel) button is selected, or 3 if the Choice 3 (focus) button is chosen.

**Example 2**

```
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", focus
enddialog
print "Choice was "; choice
```

In this example, `choice` has a value of 1 if the Choice 1 (cancel) button is chosen, 2 if the Choice 2 (ok) button is selected, or 3 if the Choice 3 (focus) button is chosen.

In both of these examples, the buttons are displayed in the same locations in the dialog box.

**See Also** `dialogbox...enddialog`



## chr (function)

Use `chr` to get a single character string defined by an ASCII value.

**Format** `x$ = chr(integer)`

**Comments** `chr` returns a one-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.

**Example 1** `cr = chr(13)`

In this example, the variable `cr` is set to ASCII value 13, which is a carriage return.

**Example 2** `reply chr(3)`

In this example, the ASCII value 3 is sent to the host.

## cksum (function)

Use `cksum` to get an integer checksum for a string of characters.

### Format

```
x = cksum(string)
```

### Comments

`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.

### Example 1

```
check = cksum(what_we_got)
```

In this example, the checksum value of the `what_we_got` variable is stored in the `check` variable.

### Example 2

```
if cksum(data_in) <> cksum(data_out) then alarm
```

In this example, an alarm sounds if the checksum of the `data_in` variable is not the same as the checksum of the `data_out` variable.

### See Also

`crc`

## class (function)

Use `class` to get the Accessory Manager class value for a single-character string.

### Format

```
x = class(string)
```

### Comments

`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). The following table lists class groupings.

Hexadecimal	Decimal	Class Contents
0x01	1	White space (space, tab, CR, LF, FF, BS, null)
0x02	2	Uppercase alpha (A–Z)
0x04	4	Lowercase alpha (a–z)
0x08	8	Legal identifier (\$, %, _)
0x10	16	Decimal digit (0–9)
0x20	32	Hexadecimal digit (A–F, a–f)
0x40	64	Delimiters: space, comma, period, tab, (, /, \, :, ;, <, =, >, !
0x80	128	Punctuation: !\, :-@, [-^, {-~

A character may fall into more than one class. For example, the comma is both a delimiter and a punctuation mark, and returns a `class` value of 0xC0 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]
```

### Comments

If no option is specified, the entire session window is cleared, and the cursor moves to the upper-left corner of the window. If an option is specified, the cursor remains in place. The following table explains the options.

<b>This option</b>	<b>Clears this</b>
<code>window</code>	The entire window
<code>line</code>	The line on which the cursor is located
<code>eow</code>	From the cursor to the end of the window
<code>bow</code>	From the cursor to the beginning of the window
<code>eol</code>	From the cursor to the end of the current line
<code>bol</code>	From the cursor to the beginning of the current line

### Example 1

```
clear bow
```

In this example, the macro clears the session window from the cursor to the beginning of the window.

### Example 2

```
clear window
```

In this example, the macro clears the entire session window.

## close (statement)

Use `close` to close an open data file.

**Format** `close [# filenum]`

**Comments** `close` ends access to an open file. If a *filenum* is not specified, all open files are closed. (All open files are closed automatically when the macro that opened them terminates.)

The # symbol must precede the file number.

**Example** `close`

In this example, all open files are closed.

**See Also** `open`

## **cls (statement)**

The `cls` statement, which is a synonym for the `clear` statement, is supported only for backward compatibility. Refer to “[clear \(statement\)](#)” on page 142.

## compile (statement)

Use `compile` to compile a macro file.

**Format** `compile "filename"`

**Comments** This statement compiles the specified macro. The compiled macro file is saved in the same directory where the source macro is found.

**Example** `compile "MENU"`

In this example, the macro called MENU is compiled.

## connected (function)

The `connected` function, which is a synonym for the `online` function, is supported only for backward compatibility. Refer to “[online \(function\)](#)” on page 263.



## copy (statement)

Use `copy` to copy a file or group of files.

### Format

```
copy [some] filefrom, fileto
```

*filefrom* must be a legal file name (full path names and wild cards are permitted). *fileto* specifies the new file name for the copy; it defaults to the current directory.

If you specify `some`, the user must approve each file before it is copied.

### Example 1

```
copy "menu.xts", "menu2.xts"
```

In this example, MENU.XTS is copied to MENU2.XTS.

### Example 2

```
copy "*.xts", "*.bak"
```

In this example, the macro copies each file with the .XTS extension and gives the copied files a .BAK extension.

### Example 3

```
copy some "*.xts", "A:"
```

In this example, the macro 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)`

**Comments** `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 macro must take some future action.

This function is case-sensitive.

**See Also** `instr`

**Example 1** `x = count("sassafras", "s")`

In this example, `count` returns the number of times the letter `s` occurs in the string "sassafras". The result is 4.

**Example 2** `x = count("sassafras", "sa")`

In this example, `count` returns the number of times the letters `s` and `a` occur in the string "sassafras". The result is 7.

## **crc (function)**

Use `crc` to determine the cyclical redundancy check value for a string.

**Format** `x = crc(string [, integer])`

**Comments** `x` is returned as the `crc` of `string`. The `crc` starts with a value of 0 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.

**Example 1** `x = crc("AM")`

In this example, `x` is assigned the `crc` value of the string `AM`.

**Example 2** `x = crc(text_line)`

In this example, `x` is assigned the `crc` value of the `text_line` variable.

**See Also** `cksum`

## curday (function)

Use `curday` to find out the current day of the month.

**Format** `x = curday`

**Comments** `curday` returns the current day of the month. The returned value is always in the range of 1–31.

**Example 1** `x = curday`  
In this example, `x` is set to the current day of the month.

**Example 2** `if curday = 15 then gosub pay_bills`  
In this example, control passes to the subroutine `pay_bills` if the current day is day 15.

**See Also** `curmonth`, `curyear`, `date`

## curdir (function)

Use `curdir` to check the name of the current directory.

**Format** `x$ = curdir[(string)]`

**Comments** `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.

**Example 1** `where_we_are = curdir`

In this example, `curdir` stores the name of the current directory in the `where_we_are` variable.

**Example 2** `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.

**See Also** `curdrive`

## curdrive (function)

Use `curdrive` to find out the current default drive.

**Format**            `x$ = curdrive`

**Comments**        `curdrive` returns a two-character string consisting of the letter of the current drive followed by a colon.

**Example 1**        `what_we_are_on = curdrive`

In this example, `curdrive` stores the letter of the current drive in the `what_we_are_on` variable.

**Example 2**        `if curdrive > "C:" then ...`

In this example, the macro takes some action if the letter of the current drive is greater than C (such as D, E, F, and so on).

**See Also**        `curdir`, `drive`

## curhour (function)

Use `curhour` to get the current hour in a 24-hour format.

**Format**

```
x = curhour
```

**Comments**

`curhour` returns an integer value containing the current hour, in the range of 0–23.

**Example 1**

```
x = curhour
```

In this example, `curhour` sets the variable `x` to the number of the current hour.

**Example 2**

```
if curhour = 23 then chain "CALLBBS"
```

In this example, the macro chains to a macro called `CALLBBS` if `curhour` is set to 23.

**See Also**

`curminute`, `cursecond`

## curminute (function)

Use `curminute` to get the current minute.

**Format** `x = curminute`

**Comments** `curminute` returns an integer containing the current minute, in the range of 0–59.

**Example 1** `x = curminute`  
In this example, `x` is set to the current minute.

**Example 2** `if curminute = 30 then ...`  
In this example, the macro takes some action if the current minute is equal to 30.

**See Also** `curhour`, `cursecond`



## curmonth (function)

Use `curmonth` to get the number of the current month.

**Format** `x = curmonth`

**Comments** `curmonth` returns an integer value containing the current month, in the range of 1–12.

**Example 1** `x = curmonth`

In this example, `x` is set to the current month.

**Example 2** `if curmonth = 12 then ...`

In this example, the macro takes some action if the current month is 12.

**See Also** `curday`, `curyear`, `date`

## cursecond (function)

Use `cursecond` to get the current second.

**Format** `x = cursecond`

**Comments** `cursecond` returns an integer value containing the current second, in the range of 0–59.

**Example 1** `x = cursecond`

In this example, `x` is set to the current second.

**Example 2** `if cursecond = 30 then ...`

In this example, the macro takes some action if the current second is equal to 30.

**See Also** `curhour`, `curminute`

## curyear (function)

Use `curyear` to find out the current year.

**Format** `x = curyear`

**Comments** `curyear` returns an integer value containing the current year.

**Example 1** `x = curyear`

In this example, `x` is set to the current year.

**Example 2** `if curyear = 2000 then ...`

In this example, the macro takes some action if the current year is 2000.

**See Also** `curday`, `curmonth`, `date`

## date (function)

Use `date` to return a date string.

### Format

```
x = date[(integer)]
```

### Comments

If *integer* is not specified or has a value of 0, `date` returns a string containing the current system date.

The returned string uses the format specified in the Short Date Style in the Control Panel. To modify the format, click the Windows Start button, point to Settings, and click Control Panel. Double-click Regional Settings, click the Date tab, and click the desired item from the Short Date Style list box.

If *integer* is specified and has a value other than 0, it indicates the number of days that have elapsed since January 1, 1900, and `date` returns the date string for that day.

### Example 1

```
x = date(31354)
```

In this example, the macro sets `x` to 11/04/85.

### See Also

`curday`, `curmonth`, `curyear`

## definput (system variable)

Use `definput` to select a default file number for input.

### Format

```
definput = filenum
```

### Comments

*filenum* must be an integer expression. `definput` lets you specify a default file number for all file input operations that follow the `definput` declaration. `eof`, `eol`, `get`, `loc`, `read`, `read line`, and `seek`, assume the file number specified by `definput` if no explicit file number is provided.

This variable is valid only for files opened in input or random mode.

### See Also

`eof`, `eol`, `get`, `loc`, `open`, `read`, `read line`, `seek`

## defoutput (system variable)

Use `defoutput` to select a default file number for output.

**Format** `defoutput = filenum`

**Comments** *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.

**See Also** `open`, `put`, `write`, `write line`

## dehex (function)

Use `dehex` to convert an `enhex` string back to its original format.

**Format** `x$ = dehex(string)`

**Comments** `dehex` converts a *string* of ASCII characters in hexadecimal format back to a string of binary data.

Since each byte in *string* is a two-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.

**Example 1** `program_line = dehex(sendable)`

In this example, `dehex` converts the ASCII hexadecimal string `sendable` to binary and returns the result in `program_line`.

**Example 2** `spread_sheet_line = dehex(nextline)`

In this example, `dehex` returns the binary equivalent of `nextline` in `spread_sheet_line`.

**See Also** `detext`, `enhex`, `entext`

## delete (statement)

Use the delete statement to delete files from the disk.

**Format**                    `delete [noask] "filename"`

**Comments**                `delete` removes a file from the disk. *filename* must be a valid file name, which can contain a drive and directory. If *filename* contains wild cards, the user is asked to confirm the deletion of each file.

Use `noask` to suppress user intervention.

**Example 1**                `delete "script1.xws"`

In this example, the file SCRIPT1.XWS is deleted.

**Example 2**                `input f$ : delete f$`

In this example, the macro accepts the file name typed 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]])
```

### Comments

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  $1 \leq \text{start} \leq \text{length}(\text{string})$ .

If  $\text{start} + \text{length}$  is greater than  $\text{length}(\text{string})$ , the leftmost  $\text{start} - 1$  bytes are returned.

### Example

```
dog_name = delete("Fixxxdo", 3, 3)
```

In this example, the macro deletes three 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`

**Comments** `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 (`""`).

**Example** `description = "Order Input"`

In this example, the macro sets `description` to the indicated string.

**See Also** `name`

## destore (function)

Use `destore` to restore strings converted with the `enstore` function; `enstore` function back to their original form.

**Format** `x$ = destore(string)`

**Comments** `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 grave accent (`^``); however, the grave accent is discarded since it is no longer needed for protection; therefore, `^`G` becomes `^G`.

You must have created `string` with `enstore`.

**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`.

**See Also** `enstore`

## detext (function)

Use `detext` to convert an `entext` string back to its original form.

**Format** `x$ = detext(string)`

**Comments** This function works with the `entext` function to transfer 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`.

**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`.

**See Also** `entext`

## device (system variable)

Use `device` to read or set the connection tool for the current session.

**Note:** EXTRA! Enterprise for Accessory Manager sessions do not support this system variable.

### Format

```
device = string
```

### Comments

The connection tool used by T 27, UTS, and ALC sessions is ICSTOOL.

After you specify the connection tool with an `assume` statement, you can read or set variables that affect the configuration of the connection tool. For more information, refer to [Chapter 7, “Connection, Terminal, and File Transfer Tools.”](#)

### Example

```
assume device "ICSTOOL"  
print PathID
```

This example displays the name of the INFOConnect path type for the current session.

### See Also

`assume`, `protocol`, `terminal`

## 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, string, 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
```

### Comments

This statement is useful for designing a user interface for your macros. Using `dialogbox...enddialog`, you can create dialog boxes that are easy to use and work like standard dialog boxes.

You must define all variables used in a dialog box before using 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.

Unless otherwise specified, Accessory Manager defines the first letter of a button or prompt as an accelerator. You can define your own accelerator by placing an ampersand (&) in the string used for the text. The letter after the ampersand becomes the accelerator.

The Dialog Box Items table describes the elements of the `dialogbox...enddialog` syntax. The Dialog Box Options table describes the options supported by those dialog box items that include *options* in their syntax.

### Dialog Box Items

Item	Description
<i>x, y</i> (for <code>dialogbox</code> )	Pixel coordinates for the dialog box
<i>w, h</i> (for <code>dialogbox</code> )	Width and height of the dialog box
<i>caption</i>	The title of the dialog box
<code>defpushbutton</code>	The default button (it has a bold border)  Pressing Enter performs the same action as clicking this button. You would normally use <code>defpushbutton</code> for the dialog box OK button.  Any dialog box must have at least one button. If there is only one button on the dialog box, use <code>defpushbutton</code> to define it.
<i>x, y</i> (for all other items in the syntax)	Pixel coordinates for the dialog box item within the dialog box  The origin of <i>x</i> and <i>y</i> is 0,0, which is the upper-left corner of the dialog box.
<i>w, h</i> (for all other items in the syntax)	Width and height of the dialog box item  A horizontal unit is 1/4 of a system font character; a vertical unit is 1/8 of a system character font.
<i>string</i>	The text to display on the dialog box
<i>options</i>	Refer to the Dialog Box Options table
<code>pushbutton</code>	A button that the user can click (such as OK or Cancel)  For this dialog box item, the width should be the length of $(string * 4) + 10$ . The height is usually 14.
<code>ltext</code>	Left-justified text within the dialog box  The width should be 4 times the length of <i>string</i> . The height is usually 8.
<code>ctext</code>	Centered text within the dialog box  The width should be 4 times the length of <i>string</i> . The height is usually 8.

Dialog Box Items (Continued)

Item	Description
<code>rtext</code>	<p>Right-justified text within the dialog box</p> <p>The width should be 4 times the length of <i>string</i>. The height is usually 8.</p>
<code>edittext</code>	<p>A text box for user input</p> <p>Precede <code>edittext</code> with <code>ltext</code>, <code>ctext</code>, or <code>rtext</code> to display a label for the text box.</p> <p>The width of the text box should be at least four times the maximum length of the string the user may type. The height is usually 12.</p>
<code>str_restul_var</code>	<p>This returns the text typed in the edit box by the user</p>
<code>radiobutton</code>	<p>A round radio or option button that is chosen when clicked by the user</p> <p>Radio buttons normally provide users with several mutually exclusive options. The first <code>radiobutton</code> in a group must have the <code>tabstop group</code> option set, or the arrow keys might not work properly in the dialog box.</p> <p>The first dialog item after a group of <code>radiobutton</code> definitions must also have the <code>tabstop group</code> option so that the operating environment knows where one group ends and the next one begins.</p> <p>The width of a <code>radiobutton</code> is generally the length of <math>(string * 4) + 10</math>. The height is generally 10.</p>
<code>result_var</code>	<p>This item is true if the radio button or check box is selected, false if not.</p> <p>For radio buttons, you must examine <code>result_var</code> for each <code>radiobutton</code> until you find one that is set to true.</p> <p>For check boxes, <code>result_var</code> is true or false depending on whether the check box was checked or not after the user exits the dialog box.</p>
<code>checkbox</code>	<p>A square box that is checked or cleared when the user clicks it</p> <p>The width of a <code>checkbox</code> should be at least the length of <math>(string * 4) + 10</math>. The height is usually 12.</p>
<code>groupbox</code>	<p>A box for a group of dialog items yet to be defined</p> <p>Dialog item definitions for this box should follow.</p>



## Dialog Box Items (Continued)

Item	Description
<i>title</i>	The title of the group box This appears in the upper border of the group box.
<i>listbox</i>	A list box If you use <i>comma_string</i> with <i>listbox</i> , the list box displays the comma-delimited strings in <i>comma_string</i> . The width of the list box should be at least four times the length of the longest string in <i>comma_string</i> . The height should be eight times the number of items from <i>comma_string</i> that you want to display at one time. The height of the list box is limited by the height of the dialog box. If you use <i>string_array</i> with <i>listbox</i> , the list box displays an array. The width of the list box should be at least four times the length of the longest string in <i>string_array</i> . The height should be eight times the number of items from <i>string_array</i> that you want to display at one time.
<i>comma_string</i>	The items to display in the list box, separated by commas
<i>string_array</i>	The array to display in the list box The array must be single-dimensional with an alternative lower boundary of 1.
<i>int_result_var</i>	The number of the list box item selected If no item was selected, zero is returned.

*Dialog Box Options*

Option	Description
tabstop	Marks a dialog item to which you can tab using the keyboard
tabstop group	Marks the beginning or end of a group of radio buttons You normally press Tab to get to the first button in a group of radio buttons, then use the arrow keys to move from one button to the next. Pressing Tab again takes you to the next dialog item after the radio button group.
focus	Defines where to place the cursor within the dialog box If this option is not used, the focus is set at the first tab stop in the dialog box.
secret	Specifies that placeholders should be displayed for the characters entered by the user. This option is useful for entries such as passwords and applies only to <code>edittext</code> .
ok	Identifies the button to associate with Enter This option applies only to <code>defpushbutton</code> or <code>pushbutton</code> .
cancel	Identifies the button to associate with Esc This option applies only to <code>defpushbutton</code> or <code>pushbutton</code> .

When the user exits the dialog box, the variable `choice` is assigned the number of the button used to exit the dialog box. For example, if the first button is chosen, `choice` is set to 1; if the fourth button is selected, `choice` is set to 4. The macro can then check `choice` to take appropriate action. Note that no variables are updated if the user clicks Cancel.

**Example 1**

```

dialogbox 61, 20, 196, 76
  ltext 6, 4, 148, 8, "About calling " + ...
  "Administration directly ..."
  ltext 6, 24, 176, 8, "When setting up " + ...
  "Accessory Manager to call Administration"
  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 click OK when ready to continue.

**Example 2**

```

string edit$
boolean check1, check2
boolean radio1, radio2
integer list1
string items[1:8]

label SampleDialog

check1 = true  -- true shows the check box selected
check2 = true
list1 = 3      -- a 3 highlights the 3rd
               -- item in the list
radio1 = true  -- true shows 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, "Sample Dialog Box"
  groupbox 4, 18, 197, 52, "Accessory Manager"
  checkbox 12, 30, 154, 12, "Designed for " + ...
    "the Windows environment", check1, tabstop
  checkbox 12, 42, 150, 12, "Includes a " + ...
    "powerful macro language", check2, tabstop ...
  focus
  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, text boxes, and radio buttons.

**See Also**

alert, choice, passchar

## display (system variable)

Use `display` to enable or disable the display of incoming characters.

**Format** `display = option`

**Comments** `option` is one of the following:

State	Result
on	Incoming characters are displayed.
off	Incoming characters are 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 macro that is using it is running.

**Example**

```
wait for "Password:"
display = off
reply password
display = on
```

In this example, the macro 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.

**See Also** `print`

## do (statement)

Use `do` to compile and run a macro.

### Format

```
do filename [, args]
```

### Comments

Like the `chain` statement, the `do` statement invokes another macro and passes control to that macro. However, unlike the macro that uses the `chain` statement, the macro issuing the `do` statement does not terminate after it invokes the child macro. Instead, it waits until the other macro returns control.

Like `chain`, `do` compiles and runs a macro source file (.XWS) if there is no compiled version of the macro, 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 macro.

You do not have to include the macro extension, but you must include the drive and directory where the macro is located.

In the `do` statement, `args` represents an optional argument list that contains the individual arguments to be passed to the other macro. Individual arguments must be separated by commas.

When you use the `do` statement to invoke another macro, the macros can exchange variable information. To pass a variable between macros, declare the variable as `public` in the invoking macro and as `external` in the invoked macro. (For information about public and external variables, refer to [Chapter 3, “Variables, Arrays, Procedures, and Functions.”](#))

For more information about invoking other macros, refer to [Chapter 4, “Interacting with the Host, Users, and Other Macros.”](#)

### Example 1

```
do "C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32\  
SCRIPT2"
```

In this example, a macro called `SCRIPT2` is invoked as a child macro.

**Example 2**

```
do "C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32\  
SCRIPT2", "CSERVE"
```

In this example, the argument CSERVE is passed to SCRIPT2.

**See Also**

`arg`, `chain`, `compile`

## drive (statement)

Use `drive` to change the default disk drive.

**Format**            `drive string`

**Comments**        *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.

**Example 1**        `drive "A:"`

In this example, the drive is changed to A.

**Example 2**        `drive dname$`

In this example, the drive is changed to the value contained in the variable `dname$`.

**See Also**        `curdrive`



## end (statement)

Use `end` to indicate the logical end of a macro.

**Format** `end`

**Comments** `end` marks the logical end of a macro. When an `end` statement is encountered, the following occurs:

- All variables associated with that macro are discarded.
- All files opened by the macro are closed.
- Execution of the macro is terminated.
- If the macro was invoked by a parent macro, execution continues in the parent macro.

Although it is a good programming practice to have an `end` statement at the physical end of the macro source code as well as at the logical end of the source code, CASL accepts the physical end of the macro as the logical end if no `end` statement is found.

**Example** `if not online then end`

In this example, the macro ends if it is not online.

**See Also** `halt`

## enhex (function)

Use `enhex` to convert a string of binary data to a string of ASCII characters in hexadecimal format.

**Format** `x$ = enhex(string)`

**Comments** `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 two-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 communication services that allow only 7-bit transfers.

Binary data strings that have been converted with `enhex` require `dehex` to restore them to 8-bit binary format.

**Example 1** `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`.

**Example 2** `reply enhex(spread_sheet_line)`

In this example, the macro sends the result of the `enhex` conversion to the host.

**See Also** `dehex`, `detext`, `entext`

## enstore (function)

Use `enstore` to convert strings that may have embedded control characters into strings of printable ASCII characters.

**Format** `x$ = enstore(string)`

**Comments** In general, control characters are changed to caret notation (that is, a Ctrl+g (bell) character is changed to ^G). 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 grave accent ( ` ) to protect any existing carets from later interpretation.

`enstore` is useful in macro 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.

**Example 1** `password = enstore("ALE" + chr(3))`

In this example, the result of the `enstore` conversion is returned in `password`.

**Example 2** `reply enstore(line_input_by_user)`

In this example, the macro sends the result of the `enstore` conversion to the host.

**See Also** `destore`

## 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 communication services that allow only 7-bit transfers.

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.

**Example 1** `sendable = entext(program_line)`

In this example, the ASCII equivalent of the binary string `program_line` is assigned to `sendable`.

**Example 2** `reply entext(spread_sheet_line)`

In this example, `spread_sheet_line` is converted to ASCII characters and then sent to the host.

**See Also** `dehex`, `detext`, `enhex`

## environ (function)

Use `environ` to obtain the value of a DOS environment variable.

### Format

```
x$ = environ(string)
```

### Comments

`environ` returns the value of a specified operating system environment, such as the path.

*string* is not case-sensitive. A null string is returned if *string* is not found in the operating system environment.

### 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)]`

**Comments** `eof` returns `true` if the file specified in *filenum* is at the end of the file. It 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; 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.

**Example**

```
string name
while not eof
  read name
  print name
wend
end
```

This code fragment reads strings from an already open sequential file and prints them to the screen. When the end-of-file marker is reached, the `while...wend` loop is terminated, and the macro ends.

**See Also** `definput`, `get`, `read`, `seek`

## 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)]
```

### Comments

`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.

Like `eof`, `eol` indicates the status of a data file following a read operation; however, `eol` 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 was written using another application.

### 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 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.

**See Also** `definput`, `read`



## errclass (system variable)

Use `errclass` to check the type of the last error.

**Format** `x = errclass`

**Comments** `errclass` contains an integer reflecting the type of error that last occurred. It is 0 if no error has occurred.

`errclass` is not cleared when you check it. It remains unchanged until another error occurs.

**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 macro executes a file transfer command.

**See Also** `errno`, `error`, `trap`

## errno (system variable)

Use `errno` to check the specific type of the last error.

**Format** `x = errno`

**Comments** `errno` contains an integer reflecting the error number, within the `errclass`, for the error that last occurred. It is 0 if no error has occurred.

`errno` is not cleared when you check it. It remains unchanged until a different error occurs.

**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`.

**See Also** `errclass`, `error`, `trap`

## error (function)

Use `error` to check for the occurrence of an error.

### Format

```
x = error
```

### Comments

`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 macro, `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.

### Example

```
trap on
compile "zark"
trap off
if error then print "Compile failed."
```

In this example, error trapping is turned on and the macro `zark` is compiled. Then error trapping is turned off. If an error occurred, the macro prints an error message.

### See Also

`errclass`, `errno`, `trap`

## exists (function)

Use `exists` to determine whether a file or subdirectory exists.

**Format** `x = exists(string)`

**Comments** `string` must be a legal file name or subdirectory name, and can contain drive specifiers, path names, and wildcard characters.

`exists` returns `true` if the item specified in `string` exists, and `false` if it does not. This function returns `true` if the directory exists, even if it's empty.

Use `exists` only to check for files and subdirectories. It does not work for root directories.

**Example 1**

```
print exists("ACCMGR32.EXE")
```

In this example, either `true` or `false` is displayed, depending on the existence of the file `ACCMGR32.EXE`.

**Example 2**

```
if exists("C:\BIN") then
    print "BIN directory!"
```

In this example, a message is displayed if the directory `BIN` exists on the `C` drive.

**Example 3**

```
if not exists(dat_file) then goto dat_error
```

In this example, the macro branches to the label `dat_error` if the `dat_file` does not exist.

## exit (statement)

Use `exit` to exit from a procedure.

**Format**                    `exit`

**Comments**                When an `exit` statement is encountered, the procedure returns control to the statement following the one that called it.

**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.

**See Also**                    `chain`, `do`, `end`, `proc...endproc`

## false (constant)

Use `false` to set a Boolean variable to logical false.

**Format**            `x = false`

**Comments**        `false` is always logical false. Like its complement `true`, `false` exists as a way to set variables on and off. If `false` is converted to an integer, its value is 0.

**Example**            `done = false`  
`while not done`  
                  `...`  
                  `...`  
`wend`

In this example, the statements in the `while...wend` construct are repeated until `done` is `true`.

**See Also**            `off`, `on`, `true`

## filefind (function)

Use `filefind` to check a file name.

### Format

```
x$ = filefind[(string [, integer])]
```

### Comments

*string* must be a legal file specification that can include drive specifiers and path names as well as wildcard characters.

`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*. The bitmap returned is the total of the possible attributes shown in the following table:

Hexadecimal	Decimal	Attribute Meaning
0x01	1	A read-only file.
0x02	2	A hidden file. The file is excluded from directory searches.
0x04	4	A system file. The file is excluded from directory searches.
0x08	8	The volume name of a disk. <b>Note:</b> This is not supported.
0x10	16	A directory.
0x20	32	An archive bit. This bit indicates the file has been changed since it was last backed up.

**Example**

```
x = filefind("*.*)
while not null(x)
  print x
  x = filefind
wend
```

In this example, the macro displays a list of files in the current directory.



## filesize (function)

Use `filesize` to check the size of a file.

**Format** `x = filesize[(filename)]`

**Comments** If *filename* is used, `filesize` returns the size of the specified file. 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 wildcard characters.

**Example 1** `progsiz = filesize("ACCMGR32.EXE")`

In this example, the size of ACCMGR32.EXE is returned in `progsiz`.

**Example 2** `print filesize`

In this example, the macro displays the size of the file found by the most recent `filefind`.

**See Also** `filefind`

## fncheck (function)

Use `fncheck` to check the validity of a file name specification.

**Format** `x = fncheck(string)`

**Comments** `fncheck` provides a quick way to parse file names. It returns a value indicating the presence or absence of various file name parts such as the drive letter, path, name, file type extension, and wildcards. For this to work properly, *string* must be a legal file name.

The parts of the file name are determined by the punctuation found in the name. For example, if a colon is found, `fncheck` assumes that a drive letter is present. The following table lists the punctuation that is checked, the parts of the file name that are assumed as a result, and the values that are returned (hexadecimal and decimal).

Punctuation	Part	Hexadecimal Value	Decimal Value
Colon	Drive	0x01	1
Backslash	Directory	0x02	2
Period	Extension	0x04	4
Question mark	Wild card	0x08	8
Asterisk	Wild card	0x10	16

The values are added together for every part of a file name that is found.

**Example**

```
print fncheck(long_file_spec)
```

In this example, the various parts of the file name `long_file_spec` are displayed.

**See Also** `fnstrip`

## fnstrip (function)

Use `fnstrip` to return specified portions of a file name.

**Format** `x$ = fnstrip(string, specifier)`

**Comments** `fnstrip` provides a quick way to parse file names, breaking them down into component parts like the drive letter, directory, and file name.

`string` must be a legal file name and can include a drive, directory, file name, and extension, as shown in the following example:

```
C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32\
ACCMGR32.EXE
```

The parts of `string` that are returned are controlled by the value of `specifier`. Valid values for `specifier` are shown in the following table.

Hexadecimal	Decimal	Portion Returned
0x00	0	The full file name
0x01	1	The directory, file name, and extension
0x02	2	The drive, file name, and extension
0x03	3	The file name and extension
0x04	4	The drive, directory, and file name (no extension)
0x05	5	The directory and file name (no extension)
0x06	6	The drive and file name (no extension)
0x07	7	The file name only (no extension)

Add 8 to `specifier` to return the string in all uppercase characters; add 16 (decimal) to return the string in all lowercase characters.

**Example 1**

```
print fnstrip(long_file_name, 3)
```

In this example, the macro displays the file name and extension.

**Example 2**

```
progrname = fnstrip(long_file_name, 7)
```

In this example, `fnstrip` returns only the file name (no extension).

**Example 3**

```
U_Case_ProgName = fnstrip ("C:\PROGRAM FILES\ ...  
ACCMGR32\ACCMGR32.EXE", 15)
```

In this example, `fnstrip` returns the file name in uppercase characters.

**See Also**

`fncheck`

## footer (system variable)

Use `footer` to define the footer to use when printing from Accessory Manager.

**Format** `footer = string`

**Comments** `string` can be any valid string expression. You can embed special characters in the string to print the date, time, and so on.

**Example** `footer = "Date: " + date`

In this example, the word `Date:` and the current date are assigned to `footer`.

**See Also** `header`

## 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]
```

### Comments

*variable* can be any integer or real variable. You do not have to declare the variable previously, but doing so is recommended. Do *not* change the value of *variable* within the `for...next` construct; this can produce erroneous results.

*startvalue*, *endvalue*, and *stepvalue* can be any type of numeric expression. *startvalue* specifies the starting value for the counter, and *endvalue* specifies the ending value. (If you do not specify a *stepvalue*, 1 is assumed.)

The statements in the `for...next` construct are performed only under the following conditions:

- The *stepvalue* is greater than or equal to 0, and the *startvalue* is less than the *endvalue*.
- The *stepvalue* is less than zero, and the *startvalue* is greater than the *endvalue*.

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 *stepvalue* is greater than or equal to 0, and if *variable* is not greater than *endvalue*, the statements are repeated. If *stepvalue* is less than 0, and if *variable* is not less than *endvalue*, the statements are repeated.

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.

**Example 1**

```
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.

**Example 2**

```
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.

**Example 3**

```
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.

**Example 4**

```
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.

## freemem (function)

Use `freemem` to find out how much memory is available.

**Format** `x = freemem`

**Comments** `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.

**Example 1** `print freemem`

In this example, the macro displays the amount of unused memory.

**Example 2** `if freemem > 64k then ...`

In this example, the macro tests whether available memory exceeds 64 KB and then performs a certain action.



## freetrack (function)

Use `freetrack` to return the lowest unused track number for the current session.

**Format**

```
x = freetrack
```

**Comments**

`freetrack` returns the value of the next available track number. It lets you write general-purpose macros that do not require a specific track number. This is particularly valuable in a macro that might form part of several other macros.

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.

**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.

**See Also**

`track (function)`, `track (statement)`

## func...endfunc (function declaration)

Use `func...endfunc` to define and name a function.

### Format

```
func name [(type argument ...  
  [, [type argument]...)] returns type  
  ...  
  ...  
endfunc
```

### Comments

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 variables within the function cannot be referenced outside the function.

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 macro. 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 macro. 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
```

You can use a similar approach to call functions in a Windows Dynamic Link Library (DLL). For more information, refer to [“Calling DLL Functions”](#) on page 77.

Functions can be in separate files. To include an external function in a macro, use the `include` compiler directive.

#### Example 1

```
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 value returned by the function depends 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`, `y` is the return value.

#### Example 2

```
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 preceding 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
```

#### See Also

`include`, `proc...endproc`

## genlabels (compiler directive)

Use `genlabels` to include or exclude label information in a compiled macro.

**Format** `genlabels option`

**Comments** `option` is one of the following:

Value	Result
off	The macro compiler suppresses label information in the compiled macro. The resulting macro is usually smaller if you use this directive.
on	The macro compiler does not suppress label information in the compiled macro. The default for the directive is on.

**Note:** You cannot use the `goto @expression` statement if your macro contains the `genlabels off` compiler directive.

**Example** `genlabels off`

In this example, `genlabels` is set to `off`.

**See Also** `genlines`

## genlines (compiler directive)

Use `genlines` to include or exclude line information in a compiled macro.

**Format** `genlines option`

**Comments** `option` is one of the following:

Value	Result
off	The macro compiler excludes line information from the compiled macro.
on	The macro compiler includes line information from the compiled macro. The default for the directive is on.

**Example** `genlines off`

In this example, `genlines` is set to `off`.

**See Also** `genlabels`, `trace`

## get (statement)

Use `get` to read characters from a random file.

**Format** `get [# filenum, ] integer, stringvar`

**Comments** `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 macro uses the value in `definput`.

If the end-of-file marker is reached during the read, *stringvar* might contain fewer than *integer* bytes, and might even be null.

Each `get` advances the file I/O pointer by *integer* positions or to the end-of-file marker, whichever comes first.

To use the `get` statement, you must open the file in random mode and have already declared *stringvar*.

**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 one byte at a time and calls a procedure to process the byte. This continues to happen until the end-of-file marker is reached.

**See Also** `definput`, `open`, `put`, `seek`

## go (statement)

Use `go` to establish communications with the host.

### Format

`go`

### Comments

`go` establishes a connection to the host and runs a session startup macro (if the session uses a session startup macro).

To determine whether the session uses a session startup macro, open the session, click Session Preferences from the Options menu, and click the CASL Macro tab. Any macro specified in the File Name text box is the session startup macro.

If the session is already connected to the host, `go` does nothing.

### See Also

`bye`, `quit`

## 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
```

### Comments

*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 macro. 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 eight nested subroutines.

When a `gosub` statement is encountered, the macro 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 macro, but it is a good programming practice to put all of your subroutines together, usually at the end of the macro.

### 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.

### See Also

`goto`, `label`, `pop`



## goto (statement)

Use `goto` to branch to a label or expression.

### Format

```
goto label  
  
or  
  
goto @expression
```

### Comments

*label* must be the name of a program label.

*expression* can be any string expression that represents a label in the macro. 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 macro, the macro branches to *label*.

**Note:** If you use the `goto @expression` form of this statement, you cannot use the `genlabels off` compiler directive.

### Example 1

```
goto main_menu
```

In this example, the macro branches to the label `main_menu`.

### Example 2

```
goto @"handle_" + xvi_keyword
```

In this example, the macro branches to the specified expression.

### See Also

```
gosub...return, label
```

## grab (statement)

Use `grab` to send the contents of the session window to a file.

**Format** `grab`

**Comments** `grab` places the text in the session window into the file specified on the Print Screen Options dialog box.

For this statement to work, Print To File must be selected on the Print Screen Printer Settings dialog box. To do this, open a session, click Print Screen from the File menu, select Print To File, and click OK.

By default, the file name is the first five letters of the session name, followed by a letter for the month (January is A, February is B, and so forth), and the day of the month. The file extension is .TXT. For example, if the session name is TCPA\_1 and the file is created on April 15, the file name is TCPA\_D15.TXT. To change the file name, click Print Screen from the File menu, make sure Print To File is selected, click Options, clear Auto Name The File, and type the desired file name in the File Name text box. (You can also click Browse and select the desired file from a list of available files.)

**Example** `grab`

**See Also** `capture`, `printer`

## halt (statement)

Use halt to stop macro execution.

**Format** halt

**Comments** When a halt statement is encountered, the macro stops immediately. If there is a related parent macro, it terminates also.

**Note:** To stop a running macro using Accessory Manager, click Stop CASL Macro from the Tools menu.

**Example** if not online then halt

In this example, the macro stops executing if the session is not connected to the host.

**See Also** end

## header (system variable)

Use `header` to define the header to use when printing from Accessory Manager.

**Format**                    `header = string`

**Comments**                `string` can be a any valid string expression. You can embed special characters in the string to print the date, time, and so on.

**Example**                    `header = "Printed using the " + description ...`  
                                  `+ " session."`

In this example, the specified string is assigned to `header`.

**See Also**                    `footer`

## hex (function)

Use `hex` to convert an integer to a hexadecimal string.

**Format** `x$ = hex(integer)`

**Comments** `hex` returns a string giving the hexadecimal representation of *integer*. If *integer* is between 0 and 65,535, the string is 4 characters long; otherwise, it is 8 characters long.

**Example**

```
print hex(32767)
```

In this example, the macro displays the hexadecimal equivalent of the integer 32,767.

## hide (statement)

Use `hide` to minimize the session window.

**Format**                    `hide`

**Comments**                This statement minimizes the session window. To minimize the Accessory Manager application window, use the `minimize` statement.

**Example**                    `hide`

**See Also**                    `minimize`, `show`, `zoom`

## hideallquickpads (statement)

The `hideallquickpads` statement is supported only for backward compatibility. Refer to “[unloadallquickpads \(statement\)](#)” on page 330.

## hidequickpad (statement)

The `hidequickpad` statement is supported only for backward compatibility. Refer to “[unloadquickpad \(statement\)](#)” on page 331.



## hms (function)

Use `hms` to return a string in a time format.

**Format** `x$ = hms(integer [, time_type])`

**Comments** `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 *systemtime*.

*time\_type* is a value that controls the format returned. It defaults to 0. The following table shows valid values for *time\_type* and the resulting time format:

Hexadecimal	Decimal	300011 Format	101 Format
0x00	0	8:20:01.1	0:00:10.1
0x01	1	8:20:01.1	10.1
0x02	2	8:20:01	0:00:10
0x03	3	8:20:01	10
0x04	4	8h20m1.1s	0h0m10.1s
0x05	5	8h20m1.1s	10.1s
0x06	6	8h20m1s	0h0m10s
0x07	7	8h20m1s	10s

**Example 1** `print hms(300011)`

In this example, the macro displays the time.

**Example 2** `print hms(systemtime, 6)`

In this example, the macro displays the number of ticks that Accessory Manager has been active in the 0h0m0s format.

**See Also** `systemtime`

## homedir (system variable)

Use `homedir` to specify the drive and directory where Accessory Manager is installed.

**Format** `homedir`

**Comments** This is a read-only string variable. You can use it as an argument for another function or statement, or you can assign the value of `homedir` to a variable you create.

**Example 1** `chdir homedir`

In this example, the macro changes the active directory to the Accessory Manager directory.

**Example 2** `run "winhelp.exe " + homedir + "\accmgr32.hlp"`

In this example, the value of `homedir` is concatenated with the strings before and after it.

**Example 3** `mydir = homedir`

In this example, the value of `homedir` is assigned to another variable. Because the new variable is not read-only, you can manipulate its value.

**See Also** `chdir`, `curdir`

## 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]
```

### Comments

*expression* can be any type of numeric, string, or Boolean expression. It can also 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 do not have to be explicitly compared to 0, but strings must be compared to produce a `true/false` value.

For example, the following values evaluate logically to `true`:

```
1
1 = 1
1 = (2-1)
"X" = "X"
"X" = upcase("x")
```

The following conditions evaluate to `false`:

```
0
1 - 1
1 = 2
"X" = "Y"
```

`then` specifies the statement to perform if *expression* 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 *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 to make the text more readable, use braces (`{ }`) to enclose a series of statements.

**Example 1**

```
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.

**Example 2**

```
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.

**Example 3**

```
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.

**Example 4**

```
if track(1) then
{
  bye
  wait 8 minutes
  print "Eight minutes have elapsed."
  end
}
```

This example shows how to use braces to indicate a series of statements in an if...then construct. This can make if...then statements easier to read.

**Example 5**

```
if x then { if y then a } else b
```

This example shows how to use braces to denote the then with which an else should be associated.

## include (compiler directive)

Use `include` to include an external file in your macro.

**Format** `include "filename"`

**Comments** `include` is a compile-time directive. It is normally used to include a source file of commonly used procedures and subroutines in a macro.

*filename* is required and must be the name of an existing file containing CASL language elements. If a file extension is omitted, `.XWS` is assumed.

`include` does not include the same file more than once during compilation.

**Example** `include "myprocs"`

In this example, the file `MYPROCS.XWS` is included in the macro.

**See Also** `chain`, `do`, `func...endfunc`, `proc...endproc`

## inject (function)

Use `inject` to return a string with some characters changed.

**Format** `x$ = inject("old_string", "repl_string" ...  
[, integer])`

**Comments** `inject` creates a new character string by 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` cannot be null. If `repl_string` is too long, it is truncated. `integer` must be in the range of  $1 \leq integer \leq$  length of `old_string`. If `integer` is omitted, the first character position is assumed.

**Example 1** `print inject("ACTMGR32.EXE", "C", 3)`

In this example, the *T* in ACTMGR32.EXE is changed to a *C* and the result is displayed.

**Example 2** `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`.

**See Also** `insert`

## inkey (function)

Use `inkey` to return the value of a keystroke.

### Format

```
x = inkey
```

### Comments

`inkey` tests for keystrokes without stopping the macro to wait for a keystroke. This is 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 value for the arrow keys, function keys, and special purpose keys (shown in the following table):

Keyboard Key	Value
F1–F10	1025–1034
Shift+F1–Shift+F10	1035–1044
Ctrl+F1–Ctrl+F10	1045–1054
Alt+F1–Alt+F10	1055–1064
Up Arrow	1281
Down Arrow	1282
Left Arrow	1283
Right Arrow	1284
Home	1285
End	1286
Page Up	1287
Page Down	1288
Insert	1297
Delete	1298

If no keystroke is waiting, `inkey` returns 0.

To clear the keyboard buffer before testing for a keystroke, use the following code:

```
while inkey : wend
```

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 Esc so the macro can continue, use the following code:

```
print at 0, 0 , "Press Esc";
while inkey <> 27
wend
```

**Example 1**

```
if inkey then end
```

In this example, the macro ends if any key is pressed.

**Example 2**

```
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 Esc is not pressed.

**See Also**

input, stroke



## input (statement)

Use `input` to accept input from the keyboard.

**Format** `input variable`

**Comments** `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 macro.

**See Also** `inkey`

## inscript (function)

Use `inscript` to check the labels in a macro.

**Format** `x = inscript(expression)`

**Comments** `inscript` uses *expression* to check for the presence of a particular label in a macro. The value returned is `true` if *expression* is a label in the currently running macro, `false` if it is not. *expression* must be a string.

**Note:** The `genlabels` compiler directive must be on for this function to work properly.

**Example** `if inscript("HA_" + user_input) then ...`

In this example, the macro tests for the presence of the specified label.

**See Also** `enlabels`, `label`

## insert (function)

Use `insert` to return a string with some characters added.

**Format**

```
x$ = insert("old_string", "insert_string" ...  
        [, integer])
```

**Comments**

`insert` creates a new character string by adding the characters in `insert_string` at the `integer` character position in `old_string`. The length of the resulting string is the combined length of `old_string` and `insert_string`.

`old_string` cannot be null. `integer` must be in the range of  $1 \leq integer \leq \text{length of } old\_string$ . If `integer` is omitted, the first character position is assumed.

**Example 1**

```
print insert("ACMGR32.EXE", "C", 2)
```

In this example, the macro inserts a *C* in the second position of ACMGR32.EXE and displays the result.

**Example 2**

```
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`.

**See Also**

`inject`

## instr (function)

Use `instr` to return the position of a substring within a string.

### Format

```
x = instr(string, sub_string [, integer])
```

### Comments

`instr` reports the position of `sub_string` in `string` starting its search at character `integer`. If `integer` is omitted, the search begins at the first character. If `sub_string` is not found within `string`, 0 is returned.

`instr` can be used within a loop to detect the presence of a character that 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
```

### Example 1

```
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`.

### Example 2

```
if instr(fname, ".") = 0 then
    fname = fname + ".XWS"
```

In this example, the macro 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)`

**Comments** `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 macro language is quite flexible as to the number base (decimal or hexadecimal) used; end `string` with an `h` if it is hexadecimal, or `k` if it is decimal. (`k` is for kilobytes, 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` 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 hyphen (`-`).

**Example** `num = intval(user_input_string)`

In this example, `user_input_string` is converted to an integer and returned in `num`.

**See Also** `str`, `val`

## jump (statement)

The jump statement, which is a synonym for the `goto` statement, is supported only for backward compatibility. For more information, refer to “[goto \(statement\)](#)” on page 211.

## keys (system variable)

Use `keys` to read or set the keyboard map for the current session.

### Format

```
keys = string
```

### Comments

`keys` specifies the name of keyboard map for the current session. You have to specify the full DOS path (drive and directory) where the file is located, as well as the `.EKM` file extension.

### Example 1

```
keys = "C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32\  
HSW.EKM"
```

In this example, the keyboard map for the session is changed to `HSW.EKM`.

### Example 2

```
if keys = "C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32\  
HSW.EKM" then ...
```

In this example, the macro performs some action if the keyboard map for the current session is `HSW.EKM`.

## label (statement)

Use `label` to specify a named reference point in a macro file.

**Format** `label labelname`

**Comments** *labelname* can be almost any printable characters. (Do not use reserved words or special characters as a label name.)

Labels are used in macros to provide a means of identifying a particular line in a program.

**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.

**See Also** `gosub...return`, `goto`



## left (function)

Use `left` to return the left portion of a string.

**Format** `x$ = left(string [, integer])`

**Comments** `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.

**Example 1** `dog_name = left("Fidox", 4)`

In this example, `left` returns `Fido`.

**Example 2** `print left(long_string, 78)`

In this example, the first 78 characters of `long_string` are displayed.

**Example 3** `reply left(dat_rec, 24)`

In this example, the first 24 characters of `dat_rec` are sent to the host.

**See Also** `mid`, `right`, `slice`, `strip`, `subst`

## length (function)

Use `length` to return the length of a string.

**Format** `x = length(string)`

**Comments** Since CASL allows strings of up to 32,767 characters, `length` always returns integers in the range of  $0 \leq \text{length of } string \leq 32767$ . `length` returns 0 if *string* is null.

**Example 1**

```
print length(dog_name), dog_name
```

In this example, the macro displays both the length of the string `dog_name` and the contents of the string.

**Example 2**

```
if length(txt_ln) then reply txt_ln
else reply "-"
```

In this example, the macro sends the contents of `txt_ln` to the host if `txt_ln` contains data. Otherwise, the macro sends a dash to the host.

## loadquickpad (statement)

Use `loadquickpad` to open and display a QuickPad.

**Format** `loadquickpad string`

**Comments** This statement loads the QuickPad specified in *string*. You do not have to include the `.EQP` file extension.

**Example**

```
if online then
  loadquickpad "apad"
```

In this example, the QuickPad named `APAD.EQP` is loaded if the session is connected to a host.

**See Also** `hideallquickpads`, `hidequickpad`, `showallquickpads`, `showquickpad`, `unloadallquickpads`, `unloadquickpad`

## loc (function)

Use `loc` to return the position of the file pointer.

**Format** `x = loc[(filenum)]`

**Comments** `loc` returns the byte position of the next read or write in a random file.

If *filenum* is omitted, the default file number is assumed. You can set the default file number using the `definput` system variable.

This function is valid only for files opened in random mode.

**Example 1** `print loc(1)`

In this example, the macro displays the location of the input/output pointer for file number 1.

**Example 2** `if loc(1) = 8k then print "Eight kilobytes read."`

In this example, the macro prints the specified phrase if the file pointer is 8 KB into the file.

**See Also** `definput`, `open`, `seek`

## lowercase (function)

Use `lowercase` to convert a string to lowercase letters.

**Format** `x$ = lowercase(string)`

**Comments** `lowercase` converts only the letters A–Z to lowercase characters. Numerals, punctuation marks, and notational symbols are unaffected.

`lowercase` is useful for testing string equivalence since it makes the string case-insensitive.

**Example 1**

```
print "Can't find "; lowercase(fl_name)
```

In this example, the macro displays a phrase that contains a file name in lowercase letters.

**Example 2**

```
if lowercase(password) = "secret" then ...
```

In this example, the macro takes some action if the contents of `password` is `secret`.

**See Also** `uppercase`

## lprint (statement)

Use `lprint` to send text to a printer.

### Format

```
lprint [item] [{ , | ; } [item]] ... [ ; ]
```

### Comments

`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. 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 macro ends without executing an `lprint EOJ`, the macro processor executes one for you. If `item` is omitted, a blank line is printed.

If the items in the list are separated by semicolons, they are printed with no space between them. If they are separated by commas, they are printed at the next tab position.

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.

### Example 1

```
lprint "This is being sent to the printer."
```

This example shows how to print a simple phrase.

### Example 2

```
lprint "There's no carriage return after this.";
```

This example shows how to suppress a carriage return.

### Example 3

```
lprint "Current protocol is " ; protocol
```

This example shows how to print two phrases with no space between them.

### Example 4

```
lprint "Hello, " , name$
```

This example shows how to print a phrase followed by an automatic tab to `name$`.

### See Also

`print`

## match (system variable)

Use `match` to check the string found during the last `wait` or `watch` statement.

**Format**                    `x$ = match`

**Comments**                `match` returns the most recent string for which the macro 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 the session is online.

**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 macro waits up to one minute for the host to send a prompt. The macro then uses the `case...endcase` construct to determine what response to send to the host.

**See Also**                    `wait`, `watch...endwatch`

## max (function)

Use `max` to return the greater of two numbers.

**Format** `x = max(number1, number2)`

**Comments** `max` compares two numbers and returns the greater of the two.

**Example**

```
integer a, b, c
a = 1
b = 2
c = max(a, b)
```

In this example, the macro 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`.

**See Also** `min`



## maximize (statement)

Use `maximize` to enlarge the Accessory Manager application window to full screen size.

**Format** `maximize`

**Comments** `maximize` lets you maximize the Accessory Manager application window. To maximize a session window, use the `zoom` statement.

**Example** `maximize`

**See Also** `minize`, `move`, `restore`, `size`, `zoom`

## mid (function)

Use `mid` to return the middle portion of a string.

**Format** `x$ = mid(string, start [, len])`

**Comments** `mid` returns the middle portion of *string* beginning at *start*, and returns *len* bytes. If *len* is omitted, or if *start* plus *len* is greater than the length of *string*, then the rest of the string is returned.

**Example 1** `dog_name = mid("Here, Fido, here boy!", 7, 4)`

In this example, `mid` returns `Fido` in `dog_name`.

**Example 2** `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.

**See Also** `left`, `right`, `slice`, `strip`, `subst`

## min (function)

Use `min` to return the lesser of two numbers.

### Format

```
x = min(number1, number2)
```

### Comments

`min` compares two numbers and returns the lesser of the two.

### Example

```
integer a, b, c
a = 1
b = 2
c = min(a, b)
```

In this example, the macro 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`.

### See Also

`max`

## minimize (statement)

Use `minimize` to reduce the Accessory Manager application window to an icon.

**Format** `minimize`

**Comments** `minimize` lets you minimize the Accessory Manager application window. To minimize a session window, use the `hide` statement.

**Example** `minimize`

**See Also** `hide`, `maximize`, `move`, `restore`, `size`

## mkdir (statement)

Use `mkdir` to create a new subdirectory.

**Format** `mkdir directory`

**Comments** *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 specified for the directory already exists.

You can also use the abbreviation `md` for this statement.

**Example 1**

```
mkdir "C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32\
FILE"
```

In this example, the macro creates a directory called `FILE` in the `C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32` directory.

**Example 2**

```
mkdir "FILE"
```

In this example, the macro creates a subdirectory called `FILE` under the current drive and directory.

**See Also** `rmdir`

## mkint (function)

Use `mkint` to convert strings to integers.

### Format

```
x = mkint(string)
```

### Comments

Use `mkstr` to convert 32-bit integers into 4-byte strings for compact storage in a file. When you read the file, use `mkint` to convert the strings to integers.

### Example

```
get #1, 4, a_string : a_num = mkint(a_string)
```

In this example, the `get` statement reads four 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`.

### See Also

`mkstr`

## mkstr (function)

Use `mkstr` to convert integers to strings for more compact file storage.

**Format** `x$ = mkstr(integer)`

**Comments** Use `mkstr` to convert 32-bit integers into 4-byte strings for compact storage in a file. When you read the file, use `mkint` to convert the strings to integers.

**Example 1**

```
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.

**Example 2**

```
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.

**See Also** `mkint`

## move (statement)

Use `move` to move the Accessory Manager application window to a new location on the screen.

**Format**                    `move x, y`

**Comments**                This statement moves the upper-left corner of the Accessory Manager application window to the location specified by `x` and `y`. `x` and `y` are the pixel coordinates of the columns and rows on the screen. The range of coordinates depends on the video hardware used.

**Example 1**                `move 2, 30`

This example shows how to move the application window to column 2, row 30.

**Example 2**                `move x, y`

In this example, the macro moves the application window to the location defined by the `x` and `y` variables.

**See Also**                `maximize, minimize, restore, size`



## name (function)

Use name to get the name of the current session.

### Format

x\$ = name

### Comments

name returns the name of the current session. The name of the session is the same as the .ADP file name and appears in the title bar of the session window.

### Example

```
if name = "ansi" then go
```

In this example, if the name of the session is ANSI, the macro connects the session to the host.

## netid (system variable)

Use `netid` to read or set a network identifier for the current session.

**Format** `netid = string`

**Comments** `netid` sets and reads the network address associated with the current session. The `netid` is limited to 40 characters.

**Note:** To set this parameter using Accessory Manager, click Session Preferences from the Options menu, click the CASL Macro tab, and type the desired string in the Network ID text box.

**Example** `netid = "CIS02"`

In this example, `netid` is set to CIS02.

## new (statement)

Use `new` to create or open a session.

### Format

```
new [path\filename]
```

### Comments

*path* is the directory where the session profiles are stored.  
*filename* is the name of a session (.ADP file).

- If you omit the path and file name, the New Session Wizard runs, and you can create a new session.
- If you include the path and file name with no extension, CASL looks for a file named with the default session profile extension. (The default is .ADP.) If you have changed the default extension in Accessory Manager, CASL will look for a file named with this new extension.
- If you include the path and file name with an extension, CASL looks for a file named with the specified session profile extension.

If you include *path\filename* and you receive an error message indicating that the file could not be found, specify the drive and directory where the session is located and try again, or check Accessory Manager to determine if you are using the correct session profile extension.

**Note:** For information on changing session profile extensions, see the *INFOConnect User's Reference*.

### Example

```
new "C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32\
TCPA_1"
```

## nextchar (function)

Use `nextchar` to return the character waiting at the communication device.

**Format** `x$ = nextchar`

**Comments** `nextchar` returns the character waiting at the communication 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.

**Example 1**

```
/* 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
```

This example uses the `nextchar` and `inkey` functions to get characters from the device and the keyboard, respectively.

**Example 2**

```
nchr = nextchar : if null(nchr) then
  gosub a_label
```

In this example, the macro tests whether or not the next character is a blank; if it is, control is passed to the subroutine `a_label`.

**See Also** `inkey`, `nextline`

## nextline (statement)

Use the `nextline` statement to get a line of characters from the communication port.

### Format

```
nextline string [, time_expr [, maxsize]]
```

### Comments

`nextline` accumulates the characters that arrive at the communication port (delimited by carriage returns) and returns them in the variable *string*.

If a carriage return has not been received since the last `nextline`, the program accumulates characters until one of the following occurs:

- A carriage return is encountered.
- The amount of time specified in *time\_expr* is reached.
- *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* is the number of seconds to wait for the next carriage return or the next character. This number can be an integer or a real (floating point) number. If *time\_expr* is reached between the receipt of characters, the characters accumulated to that point are returned and macro execution continues. You can use the `timeout` system variable to determine if the value in *time\_expr* was exceeded. If *time\_expr* is omitted, `nextline` accumulates characters until a carriage return is encountered or *maxsize* characters have accumulated.

*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.

**Example 1**

```
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.

**Example 2**

```
nextline big_string, 5.5, 100  
if timeout then bye
```

In this example, `nextline` waits 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.

**See Also**

`nextchar`, `nextline (function)`, `timeout`

## nextline (function)

Use the `nextline` function to return a line of characters from the communication port.

**Format** `x$ = nextline[(delay [, maxsize])]`

**Comments** `nextline` looks for a carriage return and then returns the string of characters that have accumulated at the communication port.

If a carriage return has not been received since the last `nextline`, the characters accumulate until one of the following occurs:

- A carriage return is encountered.
- The amount of time specified in *delay* is reached.
- *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 number of seconds to wait for the next carriage return or the next character. This number can be an integer or a real (floating point) number. If *delay* is reached between the receipt of characters, the characters accumulated to that point are returned and the macro continues executing. By default, the `nextline` function waits indefinitely.

*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.

**Example 1**

```
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.

**Example 2**

```
big_string = big_string + nextline(15, 1024)
if timeout then bye
```

In this example, `nextline` waits 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 elapse between characters. The characters are accumulated in the variable `big_string`.

**See Also**

`nextchar`, `nextline (statement)`, `timeout`



## null (function)

Use `null` to determine if a string is null.

**Format**

```
x = null(string)
```

**Comments**

`null` returns `true` if *string* is null; otherwise, it returns `false`. (Null strings have no length or contents.)

The following code fragments have equivalent results 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 ...
```

**Example**

```
print null("Fido"), null("")
```

In this example, the `null` function displays `false` for `"Fido"` and `true` for `""`.

**See Also**

`length`

## octal (function)

Use `octal` to return a number as a string in octal format.

**Format** `x$ = octal(integer)`

**Comments** `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*. The following table shows possible integer ranges and the corresponding byte length.

<u>Integer Ranges</u>	<u>Byte Length</u>
0–65,535	6
65,536–2,147,483,647	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
```

### Comments

`off` is always logical false. Like its complement `on`, `off` exists as a way to set variables.

### Example

```
echo = off
```

In this example, `echo` is set to `off`.

### See Also

`false`, `on`, `true`

## on (constant)

Use `on` to set a variable to logical true.

**Format** `x = on`

**Comments** `on` is always logical true. Like its complement `off`, `on` exists as a way to set variables.

**Example** `echo = on`

In this example, the variable `echo` is set to `on`.

**See Also** `false`, `off`, `true`

## online (function)

Use `online` to determine whether a connection is successful.

**Format**                    `x = online`

**Comments**                `online` returns `true` or `false`, depending on whether the session is online to another computer. Some macro statements and functions (such as `reply`) are inappropriate unless you are online when they are executed. You can use `online` to control program flow.

**Example 1**                `while online ...`

In this example, the macro performs some task while the session is connected to the host.

**Example 2**                `if not online then new "C:\PROGRAM FILES\ATTACHMATE\  
INFOCN2K\ACCMGR32\TCPA_1"`

In this example, if the session is not online, the macro opens the specified session.

## ontime (function)

Use `ontime` to determine the number of ticks that the session has been online.

**Format** `x = ontime`

**Comments** `ontime` returns the number of ticks that the session has been online. (One tick is one tenth of a second.) You can use `ontime` to call accounting routines, random number routines, and similar routines.

`ontime` is set to zero when a connection is established and stops counting when the session is disconnected.

To determine the number of ticks that Accessory Manager has been active, use the `systemtime` function.

**Example 1** `print ontime`

In this example, the macro displays the value in `ontime`.

**Example 2** `if ontime/600 > 30 then ...`

In this example, the macro tests the result of a mathematical computation and takes some action if the result is true.

**See Also** `online`, `systemtime`

## open (statement)

Use `open` to open a disk file.

**Format** `open mode filename as #filenum`

**Comments** Before a macro can read from or write to a file, the file must be opened. `open` opens `filename` using `filenum` for the activities allowed by `mode`.

`mode` can be any of the following options:

Option	Description
random	Allows input and output to the file at any location using <code>seek</code> , <code>get</code> , <code>put</code> , and <code>loc</code> . If the file does not exist, it is created.
input	Allows read-only sequential access of an existing file using <code>read</code> for comma-delimited ASCII records and <code>read line</code> 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 <code>write</code> for comma-delimited ASCII records and <code>write line</code> 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 <code>write</code> for comma-delimited ASCII records and <code>write line</code> 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 file name. Drive and directory names are allowed, but wild cards are not.

`filenum` must be in the range `1 <= filenum <= 8`.

You can open a file in only one mode at a time.

**Example** `open random "PATCH.DAT" as #1`

In this example, the macro opens `PATCH.DAT` in random mode with a file number of 1.

**See Also** `get`, `loc`, `put`, `read`, `read line`, `seek`, `write`, `write line`

## pack (function)

Use `pack` to return a condensed string.

**Format** `x$ = pack(string [, wild [, integer]])`

**Comments** `pack` returns `string` with duplicate occurrences of the characters in `wild` compressed according to the value of `integer`.

If `wild` is omitted, it defaults to a space.

`integer` specifies how consecutive characters in `string` are treated. The following `integer` values are valid:

Value	Result
0	All consecutive characters in <code>string</code> are compressed to a single occurrence of the first character. If <code>integer</code> is omitted, 0 is the default.
1	Only identical consecutive characters in <code>string</code> are compressed.

**Example 1** `pack("aabccdd", "abc", 0)`

In this example, `pack` returns `add` because `aabccc` is compressed to the first occurrence of the first character (`a`).

**Example 2** `pack("aabccdd", "abc", 1)`

In this example, `pack` returns `abcd` because only identical consecutive characters are compressed.

**Example 3** `pack("HELLO WORLD!", "L", 1)`

In this example, `pack` returns `HELO WORLD!` 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]])
```

### Comments

`pad` 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.

`orig_str` can be any string.

`len_int` is the number of characters that the returned string should be. If `len_int` is shorter than the length of `orig_str`, `orig_str` is truncated to `len_int` characters, with the truncation occurring on the right side of the string.

`pad_str` can be any character. If `pad_str` is omitted, it defaults to a space.

The value of `where_int` indicates where to place the padding in the string (as shown in the following table.) If `where_int` is omitted, it defaults to 1.

This value	Places the pads here
1	On the right side
2	On the left side
3	On both sides, centering <code>orig_str</code> in a field <code>len_int</code> characters long

**Example 1**

```
print pad("Hi", 6); pad("Hi", 6, "-"); ...  
    pad("Hi", 4, "+", 2)
```

In this example, the first `pad` function adds four spaces to the right of `Hi` to expand the string to six characters. The second `pad` function adds four hyphens to the right of `Hi` to expand the string to six characters. The third `pad` function adds two plus signs to the left of `Hi` to expand the string to four characters.

**Example**

```
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`.

**See Also**

`strip`

## passchar (system variable)

Use `passchar` to specify the character to display in a text box on a dialog box created using `dialogbox...enddialog` and the `secret` option.

**Format**

```
passchar = char
```

**Comments**

By default, if you create a dialog box using `dialogbox...enddialog` and use `edittext` with the `secret` option, any text that you type in the resulting text box appears as asterisks on the screen.

Using `passchar`, you can specify a different character to display. For example, rather than displaying asterisks, you could display the plus sign.

**Example**

```
passchar = "+"
```

**See Also**

```
dialogbox...enddialog
```

## password (system variable)

Use `password` to read or set a password string for the current session.

**Format** `password = string`

**Comments** `password` sets or reads the password associated with the current session. The password is limited to 40 characters.

**Note:** To set this parameter using Accessory Manager, click Session Preferences from the Options menu, click the CASL Macro tab, and type the desired string in the Password text box.

**Example 1** `password = "PRIVATE"`

This example shows how to set the password.

**Example 2** `print password`

This example shows how to print the password.

**Example 3** `reply password`

This example shows how to send the password to the host.

## perform (statement)

Use `perform` to call a procedure.

**Format** `perform procedurename [arglist]`

**Comments** `perform` is an alternate method of calling a procedure. It is like a combination of a forward declaration and a call. Use it to call procedures when they are located near the end of the macro.

*procedurename* is the name of the procedure to call.

*arglist* is a list of arguments that can be passed to the procedure. *arglist* must contain the same number and types of arguments in the same order as specified in the procedure declaration. Be sure to separate the arguments with commas.

**Example** `perform some_proc`

In this example, the procedure identified by `some_proc` is called.

**See Also** `proc...endproc`

## pop (statement)

Use `pop` to remove a return address from the `gosub` return stack.

**Format** `pop`

**Comments** 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 macro becomes somewhat convoluted. Therefore, use this statement only on those occasions where it cannot be avoided.

If the return stack is empty when the `pop` statement is encountered, an error occurs.

**Example** `pop`

**See Also** `gosub...return`

## press (statement)

Use `press` to send a series of keystrokes to the terminal emulator.

**Format** `press [string [, string] ... ] [;]`

**Comments** `press` sends the string expression *string* to the emulator.

*string* can be plain text, special keystrokes (such as F1), or terminal keystrokes that vary, depending on the type of terminal that the session is emulating.

Be sure to enclose special keystrokes and terminal emulation keystrokes in angle brackets, such as <F1> and <Transmit>. You can also use the ASCII value for a keystroke, such as <8> to represent the backspace. (Additional keystroke values are listed in “[inkey \(function\)](#)” on page 225.) Characters that are not enclosed in angle brackets are treated as plain text.

To suppress a trailing carriage return, use a semicolon at the end of the statement. You usually need the semicolon with a T 27 session. Omitting the semicolon (and thus sending a carriage return after the keystroke) can cause problems. For example, if you’re sending a communication keystroke, the carriage return generates a beep to indicate that the carriage return is being canceled. If you’re sending a cursor movement keystroke, the keystroke is performed, but then the carriage return moves the cursor to the first column.

`press` differs from `reply` in that `reply` sends its output directly to the host, while `press` passes its output through the terminal emulator. `reply` does not honor any terminal keystrokes that are part of the terminal emulator; `press` does honor such keystrokes.

This statement is valid only when the session is online.

**Example 1** `keys_out = "<up><left>" : press keys_out ;`

In this example, the Up Arrow and Left Arrow keystrokes are assigned to the variable `keys_out`, which is sent using the `press` statement.

**Example 2**

```
press "AM";
```

In this example, the macro sends the string `AM` without a trailing carriage return.

**Example 3**

```
press "<8>" ;
```

In this example, the macro sends a backspace.

**See Also**

```
reply
```



## print (statement)

Use `print` to display text in a session window.

**Format** `print [item] [{ , | ; } [item]] ... [ ; ]`

**Comments** `item` is one of the following:

`{expression | at row, col}`

The keyword `at` specifies a position in the session window; if it is omitted, printing begins at the current cursor position.

`item` 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 included, 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 else on the same line, or when printing on the last line of a session window.

`print` can be abbreviated as a question mark (?).

**Example 1** `print "The current protocol is " ; protocol`

In this example, the macro prints the text `The current protocol` is followed by the name of the selected protocol.

**Example 2** `print "This is all printed on the ";  
print "same line."`

In this example, the macro prints the text on a single line.

**Example 3** `print date , time(-1)`

In this example, the macro prints the date and the current time, with the time starting at the next tab stop.

**See Also** `grab`, `printer`

## printer (system variable)

Use `printer` to send screen output to a printer.

**Format** `printer = option`

**Comments** For the `printer` system variable, `option` is one of the following:

Option	Result
on	Accessory Manager sends the stream of characters coming from the communications port to the system printer.
off	Accessory Manager does not send the stream of characters to the system printer.

The settings specified on the Capture Options and Advanced Capture Options dialog boxes within Accessory Manager determine how the printing operates. To view these dialog boxes, make sure that Show Capture Dialog When Start Capture is selected on the Global Preferences dialog box. Then click Capture from the File menu, and click Options on the Capture Printer Settings dialog box.

**Example** `printer = off`

This example shows how to turn printing off.

**See Also** `capture`, `grab`

## proc...endproc (procedure declaration)

Use `proc...endproc` to define and name a procedure.

### Format

```
proc name [takes [type] argument
           [, [type] argument]...]
  ...
  ...
endproc
```

### Comments

A procedure is a group of statements that can be predefined in a macro and later referred to by name.

*name* is the name given to the procedure. It must be unique within the macro.

*takes* is optional and introduces a list of arguments that are passed to the procedure.

*type* is optional and indicates the type of argument. The arguments are assumed to be strings unless otherwise specified.

*argument* is any argument to the procedure. Arguments are optional, and procedures can take a number of 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.

`endproc` ends the procedure. 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 variables within the procedure cannot be referenced outside the procedure.

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. However, 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 macro. 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 macro. 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
```

**Note:** You can also use the `perform` statement to call a procedure that is not yet declared.

You can use the `proc` statement to call a procedure in a Windows Dynamic Link Library (DLL). For more information, refer to [“Calling DLL Functions”](#) on page 77.

Procedures can be in separate files. To include an external procedure in a macro, use the `include` compiler directive.

### Example 1

```
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.

**Example 2**

```

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

```

**See Also**

`func...endfunc`, `exit`, `include`, `gosub...return`, `goto`, `perform`

## protocol (system variable)

Use `protocol` to set or read the file transfer protocol.

**Note:** EXTRA! Enterprise for Accessory Manager sessions do not support this system variable.

### Format

```
protocol = string
```

### Comments

`protocol` checks or changes the protocol to use for file transfers.

*string* can be one of the file transfer protocols listed in the following table:

<b>This protocol name</b>	<b>Loads this file transfer protocol</b>
CANDE	CANDE
OS2200	OS2200
MAPPER	MAPPER®
NOFT	No file transfer protocol

**Note:** You cannot change to a file transfer protocol that is not supported by the session's terminal type. For example, you can change from CANDE to NOFT, but you cannot change from CANDE to OS2200, since the former is designed for use with T 27, and the latter for use with UTS. Any changes made using this command are written to the session's .ADP file.

For more information about file transfer protocols, refer to [Chapter 7, "Connection, Terminal, and File Transfer Tools."](#)

### Example 1

```
assume protocol "CANDE"  
protocol = "CANDE"
```

In this example, the CANDE file transfer protocol is loaded.

### Example 2

```
print protocol
```

In this example, the macro prints the current protocol selection.

### See Also

`assume`, `device`, `terminal`

## put (statement)

Use `put` to write characters to a random file.

**Format** `put [#filenum, ] string`

**Comments** `put` writes *string* to the random file specified by *filenum*. The length of *string* is the number of bytes written to the file.

*filenum* must be an open random file number. If *filenum* is omitted, the file number stored in the variable `defoutput` is assumed.

Each `put` advances the file I/O pointer by the number of positions in *string*. The `put` statement does not pad *string* to a particular length. (To pad the string, you must use the `pad` function.) The `put` statement also does not add quotation marks, carriage returns, or end-of-file markers.

If the end-of-file marker is reached during the write, the file is extended.

**Example 1** `put #1, some_string`

In this example, the macro writes `some_string` to a file with a file number of 1.

**Example 2** `put #fileno1, pad(rec, rec_len)`

In this example, `rec` is padded on the right with spaces to expand the string to `rec_len` characters, and then `rec` is written to the file designated by `fileno1`.

**See Also** `defoutput`, `open`, `pad`, `seek`

## quit (statement)

Use `quit` to close a session window.

**Note:** EXTRA! Enterprise for Accessory Manager sessions do not support this statement.

**Format**

`quit`

**Comments**

`quit` closes a session window. Unlike the `terminate` statement, `quit` does not close Accessory Manager, even if you use `quit` to end the last or only active session.

**Example**

`quit`

**See Also**

`terminate`



## quote (function)

Use `quote` to return a string enclosed in quotation marks.

### Format

```
x$ = quote(string)
```

### Comments

`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 lines containing comma-delimited fields of ASCII data in a sequential file.

**Format** `read [#filenum, ] string_var_list`

**Comments** The `read` statement operates only on files opened in input mode. *filenum* must be an open input file number. If *filenum* is omitted, the default input file number 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 an end-of-line marker 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*.

**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`.

**See Also** `definput`, `open`, `read line`

## read line (statement)

Use `read line` to read lines of text from a sequential file.

**Format** `read line [#filenum, ] string_var`

**Comments** The `read line` statement operates only on files opened in input mode.

*filenum* must be an open input file number. If *filenum* is omitted, the default input file number 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 (CR/LF) 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*.

**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`.

**See Also** `definput`, `open`, `read`

## receive (statement)

Use `receive` to receive a file from the host.

**Format** `receive filename`

**Comments** `receive` tells Accessory Manager to download a file from the host. *filename* is the name of the file to download.

The way `receive` works depends on the file transfer protocol you use. For example, some protocols automatically request information from the host while other protocols require user intervention to request data.

An error occurs if the statement is executed while the session is offline.

For 3270, you must indicate the source and destination for the file.

**Example 1** `receive fname`

In this example, `receive` downloads the file with the name assigned to the `fname` variable.

**Example 2** `receive "SALES"`

In this example, `receive` downloads a file named SALES.

**Example 3** `receive test.txt C:\test\test.txt`

In this example, `receive` downloads a file named TEXT.TXT to the test folder on the C drive. Note the single space separating the source and the destination for the file.

**See Also** `online`, `send`

## rename (statement)

Use `rename` to rename a file.

**Format** `rename [some] oldname, newname`

**Comments** This statement renames a file. *oldname* must be the name of an existing file and can contain wildcards. If *some* is included, the user is prompted for verification before each file is renamed.

**Example 1** `rename "TEST.XWS", "MAIL.XWS"`

In this example, the macro renames the existing file TEST.XWS to MAIL.XWS.

**Example 2** `rename FNAME1, FNAME2`

In this example, the macro 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
```

**Comments**

`repeat` lets you repeat a group of statements until some condition occurs. `until` specifies the condition that ends 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 repeats 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.

**Example 1**

```
x = 0
repeat
  x = x + 1
  print x
until x = 100
```

In this example, the macro prints numbers from 1 to 100.

**Example 2**

```
string guess
print "Guess how to get out of here:"
repeat
  input guess
until guess = "Good Bye!"
```

This example shows how a macro can prompt the user to type a string and repeat the prompt until the correct string (Good Bye!) is typed.

**See Also**

`while...wend`

## reply (statement)

Use `reply` to send a string of text to the communication device.

**Format** `reply [string [, string] ... ] [ ; ]`

**Comments** `reply` sends one or more strings of text directly to the communication device. *string* is a string expression containing the text to be transmitted.

`reply` sends a carriage return after it sends *string*. To suppress this, include a semicolon at the end of the statement. If you use `reply` without a string, it sends only a carriage return. You usually need the semicolon with T 27 sessions.

Use this statement only when the session is online.

For related information, see the `press` statement.

**Example 1** `reply "Hello!"`

In this example, the macro sends `Hello!`

**Example 2** `reply userid + " " + password`

or

`reply userid, " ", password`

or

```
reply userid;
reply " ";
reply password
```

In this example, the macro sends the user ID, a space, and the password.

**Example 3** `reply chr(3);`

In this example, the macro sends a `^C` to the host.

**See Also** `press`

## **request (statement)**

The `request` statement, which is a synonym for the `receive` statement, is supported only for backward compatibility. Refer to “`receive (statement)`” on page 286.



## restore (statement)

Use `restore` to restore the Accessory Manager application window to its previous size.

**Format** `restore`

**Comments** The `restore` statement restores the Accessory Manager application window to the size it was before it was maximized or minimized.

This statement applies only to the Accessory Manager application window. To restore a session window, use the `show` statement.

**Example** `restore`

**See Also** `maximize`, `minimize`, `move`, `show`, `size`

## return (statement)

Use `return` to exit a function or to return from a subroutine.

**Format**                    `return [expression]`

**Comments**                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.

**Example 1**

```
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 two numbers to determine which is larger and returns that number.

**Example 2**

```
integer i
gosub count_to_10
end
label count_to_10
    for i = 1 to 10
        print i
    next
return
```

In this example, the macro calls a subroutine to display the numbers 1 to 10. Note that the `return` statement does not return a value in this example.

**See Also**                    `func...endfunc`, `gosub...return`

## right (function)

Use `right` to return the right portion of a string.

**Format** `x$ = right(string [, integer])`

**Comments** `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.

**Example 1** `dog_name = right("Hey, Fido", 4)`

In this example, `right` returns `Fido` in `dog_name`.

**Example 2** `print right(long_string, 78)`

In this example, the last 78 characters in `long_string` are printed on the screen.

**See Also** `left`, `mid`, `slice`, `strip`, `subst`

## **rmdir (statement)**

Use `rmdir` to remove a subdirectory.

**Format** `rmdir directory`

**Comments** *directory* must be a string expression containing a valid directory name. If the directory name exists and contains no files or subdirectories, it is removed. If it does not exist or if it contains files or subdirectories, an error occurs.

You can also use the abbreviation `rd` for this statement.

**Example 1** `rmdir "C:\PROGRAM FILES\ATTACHMATE\INFOCN2K\ACCMGR32\  
TMP"`

In this example, the `rmdir` statement removes the `TMP` subdirectory.

**Example 2** `rmdir some_dirname`

In this example, `rmdir` removes the directory contained in `some_dirname`.

**See Also** `mkdir`

## run (statement)

Use `run` to run another application.

**Format** `run "filename"`

**Comments** This statement starts another application. *filename* is the name of the executable file.

If the file does not reside in a directory included in the PATH statement of your AUTOEXEC.BAT file, you must specify the drive and directory where the file is located.

**Example 1** `run "NOTEPAD.EXE"`

In this example, the macro runs Notepad. (In this case, the drive and directory are included in the PATH statement in the AUTOEXEC.BAT file, and are therefore not required in the run statement.)

**Example 2** `run "D:\APPS\CLOCK.EXE"`

In this example, the macro runs CLOCK.EXE, which is located in the APPS directory on drive D. In this case, the drive and directory are included in the run statement, since they are not included in the PATH statement in the AUTOEXEC.BAT file.

## save (statement)

Use `save` to save a session.

**Format** `save ["name"]`

**Comments** `name` is optional. If `name` is included, it must be a valid file name, and the session is saved using that name. You do not have to include the `.ADP` file extension. If `name` is not included, the session is saved under its current name.

**Example 1** `save`

In this example, the script saves the session using its current name.

**Example 2** `save "Source"`

In this example, the script saves the session as `SOURCE.ADP`.

## script (system variable)

Use `script` to specify the name of the session start-up macro.

**Format** `script = filename`

**Comments** `script` specifies the name of the macro to run each time you open the session. *filename* must be a valid file name; you do not have to include the `.XWC` file extension.

**Example 1** `script = "LOGON"`

In this example, the session start-up macro is set to `LOGON.XWC`.

**Example 2** `if script = "LOGON" then ...`

In this example, some action is taken if the start-up macro for the session is named `LOGON.XWC`.

**See Also** `startup`

## scriptdesc (compiler directive)

Use `scriptdesc` to specify a description for a macro.

**Format**                    `scriptdesc string`

**Comments**                `scriptdesc` defines descriptive text for a macro. *string* can be up to 40 characters in length.

**Example**                    `scriptdesc "Login macro for MARC"`

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) ]
```

### Comments

`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` (24-hour format).

### Example 1

```
print secno
```

In this example, the number of elapsed seconds since midnight are printed on the screen.

### Example 2

```
print secno (14, 2, 31)
```

In this example the macro prints the number of elapsed seconds since midnight for the time 2:02:31 P.M.

## seek (statement)

Use `seek` to move a random file input/output pointer.

**Format** `seek [#filenum, ] integer`

**Comments** `seek` moves a random file input/output pointer to character position *integer*. The next `get` or `put` action commences at that point. (The first byte in a file is character position 0.)

*filenum* must be an open input file number. If *filenum* is omitted, the default input file number stored in `definput` is assumed.

*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.

**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.

**See Also** `get`, `loc`, `open`, `put`

## send (statement)

Use `send` to transfer a file to a host.

**Format** `send filename`

**Comments** `send` initiates a file transfer to the host. *filename* is the name of the file to send, and can be a full path name.

The operation of this command depends on the file transfer protocol in use. For example, some file transfer protocols display a dialog box when you initiate a file transfer; others do not.

This statement is valid only when the session is online.

For 3270, you must indicate the source and destination for the file.

**Example 1** `send "B:\INVOICE"`

In this example, the `send` statement sends the file INVOICE from drive B on the PC to the host.

**Example 2** `send some_fname`

In this example, the `send` statement sends the file assigned to `some_fname`.

**Example 3** `send C:\test\test.txt test.txt`

In this example, the `send` statement sends the file TEXT.TXT from the test folder on the C drive. Note the single space separating the source and the destination for the file.

**See Also** `receive`

## sendbreak (statement)

Use `sendbreak` to send a break signal to the host.

**Note:** EXTRA! Enterprise for Accessory Manager sessions do not support this statement.

**Format** `sendbreak`

**Comments** 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.

This statement is valid only when a session is connected to a host.

**Example** `sendbreak`

## session (function)

Use `session` to find out the current session number.

### Format

```
x = session
```

### Comments

The `session` function returns the session number of the current session, which may or may not be the active session. The active session is the session that is currently using the keyboard or is waiting for keyboard input. The current session is the one in which the macro is running.

To determine if the session in which the macro is running is the active session, test the `session` function.

## sessname (function)

Use `sessname` to find out the name of another session.

**Format** `x$ = sessname(integer)`

**Comments** `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 which sessions are running concurrently.

**Example**

```
print sessname(1), sessno(sessname(1))
```

In this example, the macro displays the name and number of the session identified by the integer 1.

**See Also** `sessno`

## sessno (function)

Use `sessno` to find out the session number of a session.

**Format** `x = sessno [(string)]`

**Comments** `sessno` returns the number of the session whose name is *string*. You do not have to include the `.ADP` file extension. If there is no session with that name, 0 is returned. If you do not specify an argument, `sessno` returns the number of open sessions.

As with the `sessname` function, you can use `sessno` to find out which sessions are running concurrently.

**Example**

```
if sessno ("TCPA_1") then
    print "A TCPA session exists."
```

In this example, the macro displays a message if one of the currently open sessions is `TCPA_1.ADP`.

**See Also** `sessname`

## show (statement)

Use `show` to redisplay a minimized session window.

**Format** `show`

**Comments** This command redisplay a session window that was previously minimized with the `hide` statement.

To redisplay the Accessory Manager application window, use the `restore` statement.

**Example** `show`

**See Also** `hide`, `restore`, `zoom`



## showallquickpads (statement)

Use `showallquickpads` to show all of the QuickPads that are loaded for the current session.

**Format** `showallquickpads`

**Comments** This statement displays all of the QuickPads that were previously hidden.

**Note:** The QuickPads for the session must already be loaded, using the `loadquickpad` statement.

**Example** `showallquickpads`

**See Also** `hideallquickpads`, `hidequickpad`, `loadquickpad`, `showquickpad`

## showquickpad (statement)

The `showquickpad` statement is supported only for backward compatibility. Refer to “[loadquickpad \(statement\)](#)” on page 237.

## size (statement)

Use `size` to change the size of the Accessory Manager application window.

**Format** `size x, y`

**Comments** This statement changes the size of the Accessory Manager 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 range of coordinates is determined by the resolution of the video adapter and monitor in use.

**Example** `size 200, 350`

In this example, the application window is resized to be 200 pixels wide and 350 pixels high.

**See Also** `maximize`, `minimize`, `move`, `restore`

## slice (function)

Use `slice` to return portions of a string.

### Format

```
x$ = slice(string, integer ...  
        [, delin_str [, where_int]])
```

### Comments

`slice` returns portions of strings. *string* is the string that you want to work with. It is divided into substrings as delineated by *delin\_str*. For example, the string `alpha beta gamma` consists of three substrings (`alpha`, `beta`, and `gamma`) which are delimited by spaces. *delin\_str* can be a space, comma, or any other delimiter. (If *delin\_str* is omitted, a space is assumed.) You can specify more than one delimiter (for example, `;" :`).

When you use `slice`, the substring in *integer* position is returned. For example, if the string consists of three substrings and *integer* is `2`, the second substring is returned.

*where\_int* specifies where the function is to begin its analysis in *string*.

### Example 1

```
sub_string = slice("alpha beta gamma", 2)
```

In this example, `slice` returns `beta`.

### Example 2

```
print slice("alpha, beta, gamma", 2, ",")
```

In this example, `beta` is displayed on the screen.

### Example 3

```
sub_string = slice("alpha, beta gamma.delta", ...  
                 3, ",.")
```

In this example, `slice` returns `delta`.

### See Also

`left`, `mid`, `right`, `strip`, `subst`

## startup (system variable)

Use `startup` to read or set the name of a macro to run when Accessory Manager is started.

**Note:** EXTRA! Enterprise for Accessory Manager sessions do not support this system variable.

### Format

```
startup = string
```

### Comments

`startup` sets or reads the name of the macro to run automatically when you run Accessory Manager. If `startup` is null, no macro is run at start-up time.

*string* must be a valid file name. You do not have to include the .XWC file extension.

### Example 1

```
startup = "AUTOEXEC"
```

In this example, a macro called AUTOEXEC.XWC runs when Accessory Manager is started.

### Example 2

```
startup = ""
```

In this example, `startup` is null, so no macro is run when Accessory Manager is started.

### See Also

```
script
```

## str (function)

Use `str` to convert a number to string format.

**Format** `x$ = str(number)`

**Comments** `str` converts numbers to strings. *number* can be a real (floating point) number or an integer. `str` does not add any leading or trailing spaces.

**Example 1**

```
print 2 : print str(2) : print length(str(2))
```

In this example, the macro displays three 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.

**Example 2**

```
reply str(shares_to_buy)
```

In this example, the macro sends the string equivalent of `shares_to_buy` to the host.

**Example 3**

```
integer counter
string items[10]
for counter = 1 to 10
    items[counter] = "item" + str(counter)
    print items[counter]
next
```

In this example, the macro declares `counter` as an integer and `items` as an array of ten strings. The `for...next` construct is used to display the individual elements in the array.

**See Also** `intval`, `val`

## strip (function)

Use `strip` to return a string with certain characters removed.

### Format

```
x$ = strip(string [, wild [, where_int]])
```

### Comments

`strip` removes unwanted characters from strings. This function is useful for removing unwanted characters from lines read from word processing text files, leading zeros, and similar characters.

`string` is the string to work with. `wild` can be either the string of characters that you want to remove from `string` or an integer that represents the Accessory Manager character classes that you want to remove. (For a list of these integers, refer to “[class \(function\)](#)” on page 141.) The default value for `wild` is a space.

`where_int` can be one of the following:

Value	Result
0	Strip all occurrences of <code>wild</code> . This is the default.
1	Strip from the right side, stopping at the first occurrence of a character not in <code>wild</code> .
2	Strip from the left side, stopping at the first occurrence of a character not in <code>wild</code> .
3	Strip from both the right and left sides, stopping on each side at the first occurrence of a character not in <code>wild</code> .

### Example 1

```
print strip("0123456", "0", 2)
```

In this example, the macro displays 123456.

### Example 2

```
print strip("Sassafras", "as", 0)
```

In this example, the macro prints `fr`.

### Example 3

```
reply strip(strip(user_resp, junk, 0), " ", 3)
```

In this example, the macro 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.

### See Also

`left`, `mid`, `right`, `slice`, `subst`

## stroke (function)

Use `stroke` to wait for the next keystroke from the keyboard.

**Format** `x = stroke`

**Comments** `stroke` is similar to the `inkey` function, but `stroke` stops the macro 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. (Refer to “[inkey \(function\)](#)” on page 225 for a list of keys and their corresponding numbers.)

**Example**

```
print "Press a key to see its value"; : print stroke
```

In this example, the macro prints a message followed by the value of the key that is pressed.

**See Also** `inkey`



## subst (function)

Use `subst` to return a string with certain characters substituted.

**Format** `x$ = subst(string, old_str, new_str)`

**Comments** `subst` searches `string` for each occurrence of `old_str` and substitutes the characters in `new_str`.

**Example**

```
print subst("alpha", "a", "b")
```

In this example, the macro prints `blphb`.

**See Also** `left`, `mid`, `right`, `slice`, `strip`

## systemtime (function)

Use `systemtime` to return the number of ticks Accessory Manager has been active.

**Format** `x = systemtime`

**Comments** `systemtime` returns the number of ticks that Accessory Manager has been active. (One tick is one tenth of a second.) You can use `systemtime` in delay loops, random number routines, and similar routines.

To determine the number of ticks that a session has been online, use the `ontime` function.

**Example 1** `print systemtime`

In this example, the value of `systemtime` is displayed.

**Example 2** `if systemtime mod 100 = 0 then ...`

In this example, the macro takes some action if the value of `systemtime` divided by 100 is zero.

**See Also** `ontime`

## tabwidth (module variable)

Use `tabwidth` to determine the number of spaces a tab character moves the cursor.

**Format** `tabwidth = integer`

**Comments** This variable determines the number of spaces that the cursor moves when the tab character is received. *integer* can be any number from 1 to 80. The default is 8.

**Example** `tabwidth = 15`

In this example, `tabwidth` is set to 15 spaces.

## terminal (system variable)

Use `terminal` to read or set the type of the terminal emulation used by the session.

**Note:** EXTRA! Enterprise for Accessory Manager 3270 and 5250 sessions do not support this item; VT™ sessions do support it.

### Format

`terminal = string`

### Comments

`terminal` specifies the type of terminal emulation to use for the current session. `string` can be one of the following:

String	Sub-Models (use the <code>termmodel</code> variable)	Emulation Type
DCAT27	None	T 27
AMUTS	UTS20, UTS40, UTS60	UTS

**Note:** You cannot change a session from one terminal emulation type to another. For example, you cannot change a T 27 session to a UTS session. However, you can change from one sub-model to another. For example, you can change from a UTS 20 to a UTS 60 session.

For more information about terminal tools, refer to [Chapter 7, “Connection, Terminal, and File Transfer Tools.”](#)

### Example 1

```
assume terminal "AMUTS"  
terminal = "AMUTS"  
termmodel = "UTS60"
```

This example shows how to load UTS 60 terminal emulation.

### Example 2

```
print terminal
```

This example shows how to print the current terminal emulation selection.

### See Also

`assume`, `device`, `protocol`

## terminate (statement)

Use `terminate` to exit Accessory Manager.

**Format** `terminate`

**Comments** `terminate` exits Accessory Manager.

To close just a session, use the `quit` statement.

**Example**

```
clear
print "Accessory Manager will close in 5 seconds."
for i = 1 to 5
    print at 5, 5, time(-1)
    wait 1 second
next
terminate
```

In this example, the macro clears the window and then displays a message on the screen. Next, using the `for . . . next` construct, the macro displays the current time once every second until five seconds have elapsed. Finally, it closes Accessory Manager.

**See Also** `quit`

## time (function)

Use `time` to return a formatted time string.

**Format** `x$ = time(integer)`

**Comments** `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.

**Example 1**

```
print time(-1)
```

This example prints the current time.

**Example 2**

```
x = time(32431)
```

In this example, the time represented by 32,431 seconds after midnight is returned in `x`.

**Example 3**

```
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 macro is online, each line of text from the host is written to the file. Then the file is closed.

**See Also** `curhour`, `curminute`, `cursecond`

## timeout (system variable)

Use `timeout` to determine the status of the most recent `nextline`, `wait`, or `watch...endwatch` statement.

**Format** `timeout`

**Comments** `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 specified before finding the condition for which it was looking.

**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. In this case, the host wants a number of carriage returns so it can check the baud rate, parity, and stop bits. The carriage returns should be sent about once every second, and it will take an arbitrary number of carriage returns before the host returns the login prompt. When it is ready, the host sends the phrase `Login:`.

**See Also** `nextline`, `wait`, `watch...endwatch`

## trace (statement)

Use `trace` to trace how the lines in a macro are executing.

**Format** `trace option`

**Comments** `trace` can be useful for debugging macros.

`option` is one of the following:

Value	Result
on	The macro displays source macro line numbers as the statements in the macro are executed.
off	The macro does not display source macro line numbers as the statements in the macro are executed.

**Example** `trace on`

In this example, tracing is activated.

**See Also** `genlines`



## track (statement)

Use the `track` statement to watch for strings or keystrokes while online.

### Format

```
track [tracknum, ] condition
```

### Comments

`track` lets you check for any number of events or incoming strings while the macro is online, and then take some action based on which events occur.

`track` events take precedence over `wait` and `watch` events. If a `track` event occurs while a macro 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 (described below), 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 have to check the event with an `if...then...else` statement.

In the `track` statement, *tracknum* is the track number for the `track` statement. You should include *tracknum* unless the *condition* is routine *label* | *procedure* or `clear`. 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 macro that set them is still running. When the macro ends, the track numbers are closed.

*condition* is one or more of the following, separated by commas:

Condition	Result														
[ <i>case</i> ] [ <i>space</i> ] <i>string</i>	<p>When the string specified in <i>string</i> is received, the value of the corresponding track function is set to true.</p> <p><i>case</i> indicates that the case of <i>string</i> must be matched. If <i>case</i> is omitted, the case of <i>string</i> is ignored.</p> <p><i>space</i> indicates that all white-space characters in <i>string</i> (such as spaces or tabs) must be matched. If <i>space</i> is omitted, white space is ignored.</p> <p><i>string</i> can be any string or one of the following special sequences:</p> <table border="1"> <thead> <tr> <th>Sequence</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>~_ (underscore)</td> <td>Any white-space character</td> </tr> <tr> <td>~A</td> <td>Any uppercase letter</td> </tr> <tr> <td>~a</td> <td>Any lowercase letter</td> </tr> <tr> <td>~#</td> <td>Any digit (0–9)</td> </tr> <tr> <td>~X</td> <td>Any letter or digit</td> </tr> <tr> <td>~?</td> <td>Any single character</td> </tr> </tbody> </table> <p>A tilde ( ~ ) with a dash ( - ) followed by a special sequence character indicates that one or more occurrences of the sequence should be tracked. For example, ~# indicates that one or more occurrences of any digit (0–9) should be tracked.</p> <p>For this condition to work properly, the session must be online.</p>	Sequence	Meaning	~_ (underscore)	Any white-space character	~A	Any uppercase letter	~a	Any lowercase letter	~#	Any digit (0–9)	~X	Any letter or digit	~?	Any single character
Sequence	Meaning														
~_ (underscore)	Any white-space character														
~A	Any uppercase letter														
~a	Any lowercase letter														
~#	Any digit (0–9)														
~X	Any letter or digit														
~?	Any single character														
<i>quiet time</i>	<p><i>quiet</i> indicates that the macro should wait until the communication line is quiet (no characters are received) for the amount of time specified in <i>time</i>.</p> <p><i>time</i> is one of the following time expressions:</p> <ul style="list-style-type: none"> <li>■ <i>n</i> hours</li> <li>■ <i>n</i> minutes</li> <li>■ <i>n</i> seconds</li> <li>■ <i>n</i> ticks (1/10 second each)</li> </ul> <p>For this condition to work properly, the session must be online.</p>														

Condition	Result
<code>key <i>stroke_value</i></code>	<p><code>key</code> specifies a keyboard character to track.</p> <p><i>stroke_value</i> is the ASCII value (0–127) of the key pressed. For the values for special keystrokes (such as the function keys or arrow keys), refer to “<a href="#">inkey (function)</a>” on page 225. The keyboard character comes from the local keyboard, not the communication line.</p>
<code>routine <i>procedure</i></code>	<p>Use <code>track routine</code> to designate a subroutine or procedure that handles the track event.</p> <p><i>procedure</i> is the name of the subroutine or procedure.</p>
<code>clear</code>	Use <code>track clear</code> to clear all tracked items and reset all of the track flags.

To stop tracking a particular item, set the item to a null string.

You can use the match system variable to return the string found during the last track operation.

### 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 : new "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 uses both the `track` statement and the `track` function to watch for problems or Alt+A during an e-mail session.

### See Also

`freetrack`, `inkey`, `match`, `track (function)`, `wait`, `watch...endwatch`

## 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  
  
or  
  
x = track(tracknum)
```

### Comments

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 macro 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 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 (`x = track`) 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 (`x = track(tracknum)`) returns `true` if the specified track event has occurred. Checking the function clears it.

**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 macro 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.

**See Also**

`match`, `track (statement)`, `wait`, `watch...endwatch`

## trap (compiler directive)

Use `trap` to control error trapping.

**Format** `trap option`

**Comments** `trap` lets you control whether the macro continues to run when errors occur that would normally stop the macro.

*option* is one of the following:

Value	Result
<code>on</code>	An error condition does not interrupt the running of the macro.
<code>off</code>	An error condition interrupts the running of the macro. This is the default state.

When `trap` is `on`, use the error function and the `errclass` and `errno` system variables to determine the occurrence, class, and number of the 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 macro.

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 macro branches to an error-handling routine if an error occurs when the `send` statement is executed.

**See Also** `errclass`, `errno`, `error`

## true (constant)

Use `true` to set a variable to logical true.

### Format

```
x = true
```

### Comments

`true` is always logical true. Like its complement `false`, `true` exists as a way to set variables on and off. If `true` is converted to an integer, its value is 1.

### 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`.

### See Also

`false`, `off`, `on`

## unloadallquickpads (statement)

Use `unloadallquickpads` to unload all QuickPads for the current session.

**Format** `unloadallquickpads`

**Comments** This statement unloads all loaded QuickPads for the current session. To unload one specific QuickPad, use the `unloadquickpad` statement.

**Example** `unloadallquickpads`

**See Also** `hideallquickpads`, `hidequickpad`, `loadquickpad`, `showallquickpads`, `showquickpad`, `unloadquickpad`



## unloadquickpad (statement)

Use `unloadquickpad` to unload the specified QuickPad for the current session.

**Format** `unloadquickpad string`

**Comments** This statement unloads the QuickPad specified in *string*. You do not have to specify the .EQP file extension.

**Example** `unloadquickpad "apad"`

In this example, the QuickPad APAD.EQP is unloaded.

**See Also** `hideallquickpads`, `hidequickpad`, `loadquickpad`, `showallquickpads`, `showquickpad`, `unloadallquickpads`

## upcase (function)

Use `upcase` to convert a string to uppercase letters.

**Format** `x$ = upcase(string)`

**Comments** `upcase` converts only the letters a–z to uppercase characters. Numerals, punctuation marks, and notational symbols are unaffected.

**Example**

```
string yn
print "Do this again?";
input yn
if upcase(yn) = "Y" then goto start
```

In this example, the character typed by the user (which is stored in the `yn` variable) is checked to determine if it is an uppercase Y. If it is, the macro branches to the label `start`.

**See Also** `lowercase`

## userid (system variable)

Use `userid` to read or set a user number or identifier for a session.

**Format** `userid = string`

**Comments** `userid` sets or reads the user identification associated with the current session. `userid` is limited to 40 characters.

**Note:** To set this parameter using Accessory Manager, click Session Preferences from the Options menu, click the CASL Macro tab, and type the desired string in the User ID text box.

**Example 1** `userid = "76004,302"`

In this example, `userid` is set to the specified string.

**Example 2** `reply userid`

In this example, `userid` is sent to the host.

**Example 3** `userid = ""`

In this example, `userid` is cleared.

## val (function)

Use `val` to return the numeric value of a string.

### Format

```
x = val(string)
```

### Comments

Like the `intval` function, `val` 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 number. 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 0–9, ., e, E, -, and +.

### Example

```
num = val(user_input_string)
```

In this example, `user_input_string` is converted to a real number and returned in `num`.

### See Also

`intval`, `str`

## version (function)

Use `version` to return the Accessory Manager version number.

**Format** `x$ = version`

**Comments** `version` returns the Accessory Manager version number as a string.

To check the version number of Windows, use the `winversion` function.

**Example** `print version`

In this example, the Accessory Manager version number is displayed.

**See Also** `winversion`

## wait (statement)

Use `wait` to wait for a specific event to occur or to pause the macro.

**Note:** EXTRA! Enterprise for Accessory Manager sessions do not support this statement.

**Format** `wait [time] [for condition]`

**Comments** The `wait` statement waits the amount of time specified in `time` for the specified condition to occur.

`time` is one of the following time expressions:

- `n hours`
- `n minutes`
- `n seconds`
- `n ticks (1/10 second each)`

If `time` is included and the specified condition occurs within that time period, the macro resumes running.

If `time` is included and the specified condition does not occur within that time period, the `timeout` system variable returns `true`.

If `time` is omitted, the macro waits indefinitely for the specified condition to occur.

The `wait time` construct can be used whether the session is off line or online.

*condition* is one or more of the following, separated by commas:

Condition	Result														
[ <i>case</i> ] [ <i>space</i> ] <i>string</i>	<p>When the string specified in <i>string</i> is received, the macro continues.</p> <p><i>case</i> indicates that the case of <i>string</i> must be matched. If <i>case</i> is omitted, the case of <i>string</i> is ignored.</p> <p><i>space</i> indicates that all white-space characters in <i>string</i> (such as spaces or tabs) must be matched. If <i>string</i> ends with a space and you want to match that space, you must use &lt;Space&gt; in your string. If <i>space</i> is omitted, white space is ignored.</p> <p><i>string</i> can be any string or one of the following special sequences:</p> <table border="1"> <thead> <tr> <th>Sequence</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>~_ (underscore)</td> <td>Any white-space character</td> </tr> <tr> <td>~A</td> <td>Any uppercase letter</td> </tr> <tr> <td>~a</td> <td>Any lowercase letter</td> </tr> <tr> <td>~#</td> <td>Any digit (0–9)</td> </tr> <tr> <td>~X</td> <td>Any letter or digit</td> </tr> <tr> <td>~?</td> <td>Any single character</td> </tr> </tbody> </table> <p>For this condition to work properly, the session must be online.</p>	Sequence	Meaning	~_ (underscore)	Any white-space character	~A	Any uppercase letter	~a	Any lowercase letter	~#	Any digit (0–9)	~X	Any letter or digit	~?	Any single character
Sequence	Meaning														
~_ (underscore)	Any white-space character														
~A	Any uppercase letter														
~a	Any lowercase letter														
~#	Any digit (0–9)														
~X	Any letter or digit														
~?	Any single character														
<i>quiet time</i>	<p><i>quiet</i> indicates that the macro should wait until the communication line is quiet (no characters are received) for the amount of time specified in <i>time</i>.</p> <p><i>time</i> is one of the following time expressions:</p> <ul style="list-style-type: none"> <li>■ <i>n</i> hours</li> <li>■ <i>n</i> minutes</li> <li>■ <i>n</i> seconds</li> <li>■ <i>n</i> ticks (1/10 second each)</li> </ul> <p>For this condition to work properly, the session must be online.</p>														

Condition	Result
<code>key <i>stroke_value</i></code>	<p><code>key</code> specifies a keyboard character for which to wait.</p> <p><code>stroke_value</code> is the ASCII value (1–127) of the key pressed. For the values for special keystrokes (such as the function keys or arrow keys), refer to “<a href="#">inkey (function)</a>” on page 225. <code>key 0</code> causes the macro to wait for any keystroke.</p> <p>You can retrieve the value of the key that was pressed using the <code>match</code> function.</p> <p>Note that the keyboard character comes from the local keyboard, not the communication line.</p>
<code>count <i>integer</i></code>	<p><code>count</code> indicates to wait for the number of characters specified in <code>integer</code>.</p> <p>For this condition to work properly, the session must be online.</p>

When writing very long macros, you might need to add some `wait` statements to give Accessory Manager time to process the macro. To do this, add `wait 5 ticks` at several points throughout the macro.

If you have problems with the `wait for string` construct (for example, if data seems to be missing from the display), add a second `wait` statement. You can wait for a string that is not at the end of a data stream and still display the entire data stream by using two `wait` statements in sequence as follows:

```
wait for "string"
/* data up to and including string is displayed */
wait for quiet 1 tick
/* the rest of the data stream is displayed */
```

**Example 1**

```
wait for "Login:" : reply userid
```

In this example, the macro waits indefinitely for the specified phrase and sends the information stored in the `userid` system variable to the host.

**Example 2**

```
wait 1 second for "Hello"
```

In this example, the macro waits one second for the specified phrase.



**Example 3**

```
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 macro 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.

**Example 4**

```
wait 20 seconds for "in:" : if timeout then
  goto no_ans
```

In this example, the macro waits 20 seconds for a phrase. If the phrase does not arrive within 20 seconds, the macro branches to the label `no_ans`.

**Example 5**

```
wait for count 10
```

In this example, the macro waits until ten characters are received.

**Example 6**

```
wait for case "UserID:"
```

In this example, the macro waits for an exact upper- and lowercase match for the `UserID:` prompt.

**See Also**

```
inkey, match, online, timeout, track (statement),
watch...endwatch
```

## watch...endwatch (statements)

Use `watch...endwatch` to watch for one of several strings of text from the communication device or for a keystroke.

**Note:** EXTRA! Enterprise for Accessory Manager sessions do not support these statements.

### Format

```
watch [time] for
  [[case] [space] string : [statement group]]
  [quiet time] : [statement group]
  [key stroke_value] : [statement group]
  [count integer] : [statement group]
endwatch
```

### Comments

The `watch` statement waits the length of time specified in *time* for one of the specified conditions to occur and then performs the specified *statement group*.

*time* is one of the following time expressions:

- *n* hours
- *n* minutes
- *n* seconds
- *n* ticks (1/10 second each)

If *time* is included and the specified condition occurs within that time period, the specified statement group is performed, and the program logic then continues with the statement following `endwatch`.

If *time* is included and the specified condition does not occur within that time period, the `timeout` system variable returns `true`.

If *time* is omitted, the macro waits indefinitely for the specified condition to occur.

The following table explains the `watch` conditions:

Condition	Result														
<code>[case] [space] string</code>	<p>When the string specified in <code>string</code> is received, the subsequent <code>statement group</code> is performed.</p> <p><code>case</code> indicates that the case of <code>string</code> must be matched. If <code>case</code> is omitted, the case of <code>string</code> is ignored.</p> <p><code>space</code> indicates that all white-space characters in <code>string</code> (such as spaces or tabs) must be matched. If <code>space</code> is omitted, white space is ignored.</p> <p><code>string</code> can be any string or one of the following special sequences:</p> <table border="1"> <thead> <tr> <th>Sequence</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td><code>~_</code> (underscore)</td> <td>Any white-space character</td> </tr> <tr> <td><code>~A</code></td> <td>Any uppercase letter</td> </tr> <tr> <td><code>~a</code></td> <td>Any lowercase letter</td> </tr> <tr> <td><code>~#</code></td> <td>Any digit (0–9)</td> </tr> <tr> <td><code>~X</code></td> <td>Any letter or digit</td> </tr> <tr> <td><code>~?</code></td> <td>Any single character</td> </tr> </tbody> </table>	Sequence	Meaning	<code>~_</code> (underscore)	Any white-space character	<code>~A</code>	Any uppercase letter	<code>~a</code>	Any lowercase letter	<code>~#</code>	Any digit (0–9)	<code>~X</code>	Any letter or digit	<code>~?</code>	Any single character
Sequence	Meaning														
<code>~_</code> (underscore)	Any white-space character														
<code>~A</code>	Any uppercase letter														
<code>~a</code>	Any lowercase letter														
<code>~#</code>	Any digit (0–9)														
<code>~X</code>	Any letter or digit														
<code>~?</code>	Any single character														
	For this condition to work properly, the session must be online.														
<code>quiet time</code>	<p><code>quiet</code> indicates that the macro should wait until the communication line is quiet (no characters are received) for the amount of time specified in <code>time</code> before performing the <code>statement group</code>.</p> <p><code>time</code> is one of the following time expressions:</p> <ul style="list-style-type: none"> <li>■ <code>n hours</code></li> <li>■ <code>n minutes</code></li> <li>■ <code>n seconds</code></li> <li>■ <code>n ticks</code> (1/10 second each)</li> </ul> <p>For this condition to work properly, the session must be online.</p>														

Condition	Result
<code>key <i>stroke_value</i></code>	<p><code>key</code> specifies a keyboard character for which to watch.</p> <p><code>stroke_value</code> is the ASCII value (0–127) of the key pressed. For the values for special keystrokes (such as the function keys or arrow keys), refer to “<a href="#">inkey (function)</a>” on page 225.</p> <p>You can retrieve the value of the key using the <code>match</code> function.</p> <p>The keyboard character comes from the keyboard, not the communication line.</p>
<code>count <i>integer</i></code>	<p><code>count</code> indicates to watch for the number of characters specified in <code>integer</code>.</p> <p>For this condition to work properly, the session must be online.</p>

`statement group` is any CASL statement.

The `watch...endwatch` construct is not a looping construct. When one of the watch conditions is met, the macro executes the appropriate statement(s). To use these statements in a loop, place them inside a `while...wend` construct.

**Example 1**

```
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 macro watches for one of the specified events. If any of the events occurs, the statements to the right of the colon are executed, and the `watch...endwatch` construct is completed.

**Example 2**

```
while online
  watch for
    "graphics" : reply "Yes"
    "first name" : reply userid
    "password" : reply password : end
  endwatch
wend
```

This example includes the `watch...endwatch` construct in a `while...wend` loop. The `while...wend` construct continues to loop until watch receives the password: prompt.

**See Also**

`inkey`, `match`, `track`, `wait`, `while...wend`

## weekday (function)

Use `weekday` to return the number of the day of the week.

**Format** `x = weekday[(integer)]`

**Comments** `weekday` returns the number of the current day of the week. Sunday is 0, 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.

**Example**

```
print weekday, weekday(365)
```

For a Friday, the macro in this example prints 5, a tab, and 1.

**See Also** `curday`

## 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
```

### Comments

`while...wend` lets you perform one or more statements as long as a certain expression is true.

*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.

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, make sure that the terminating condition (*expression*) will eventually become true, or that there is some other exit from the loop.

### Example

```
x = 1
while x <> 100
    print x
    x = x + 1
wend
```

In this example, the macro prints the numbers 1 through 99.

### See Also

`repeat...until`

## winchar (function)

Use `winchar` to return the ASCII value of a character read from a session window.

**Format** `x = winchar(row, col)`

**Comments** `winchar` reads a character from a session window at `row, col`. The `winchar` function helps you determine the results of operations not under macro control, such as the appearance of a certain character at a certain location on the screen while under the control of a host.

**Example** `char1 = winchar(1, 1)`

In this example, the character at row 1, column 1 is stored in `char1`.

**See Also** `nextchar`, `nextline`, `winstring`

## winsize (function)

Use `winsize` to return the number of visible columns in the session window.

**Format** `x = winsize`

**Comments** `winsize` returns the number of visible columns in the session window. This function is useful in macros that display information on the screen and have to accommodate the size of the session window.

**Example 1** `print winsize`

In this example, the macro prints the number of visible columns in the session window at its current size.

**Example 2** `if winsize < 80 then zoom`

If the session window is less than 80 columns in width, this statement maximizes it.

**See Also** `winsizey`



## winsizey (function)

Use `winsizey` to return the number of visible rows in the session window.

**Format** `x = winsizey`

**Comments** `winsizey` returns the number of visible rows in the session window. This function is useful in macros that display information on the screen and have to accommodate the size of the session window.

**Example** `if winsizey < 24 then zoom`

If the session window is less than 24 rows in length, this statement maximizes it.

**See Also** `winsizex`

## winstring (function)

Use `winstring` to return a string read from a session window.

**Format** `x$ = winstring(row, col, len)`

**Comments** `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 macro control, such as the appearance of a certain string at a certain location on the screen while under the control of a host.

**Example**

```
string data
data = winstring(10, 10, 11)
if data = "Login name:" then reply userid
```

In this example, if the phrase `Login name:` appears in the session window beginning at row 10, column 10, then the `userid` system variable is sent to the host.

## winversion (function)

Use `winversion` to check the Windows version number.

**Format**                    `x$ = winversion`

**Comments**                `winversion` returns the Windows version number as a string.  
To check the version number of Accessory Manager, use the `version` function.

**Example**                    `print winversion`  
In this example, the macro displays the Windows version number on the screen.

**See Also**                    `version`

## write (statement)

Use `write` to write lines containing comma-delimited fields of ASCII data to a sequential file.

**Format** `write [#filenum, ] [item] [{, | ;} ...  
[item]] ... [;]`

**Comments** The `write` statement writes lines containing comma-delimited fields of ASCII data to a sequential file. This statement operates only on files opened in output or append modes.

*filenum* must be an open file output number. If *filenum* is omitted, the file number stored in the variable `defoutput` is assumed.

Each `write` adds the specified items to the file, with each 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 an *item* includes a comma or quotation marks, use the `quote` function to enclose the item in quotation marks.

Normally, `write` terminates each addition to the file with a carriage-return/line-feed (CR/LF). To suppress the CR/LF, use the trailing semicolon.

**Example 1**

```
open output file_name as #1
write #1, alpha, beta, gamma;
close #1
```

In this example, the macro opens a file, writes the specified strings to the file, and closes the file.

**Example 2** `write #1, quote(var1), quote(var2), quote(var3)`

In this example, the macro encloses the data strings in quotation marks before writing them to the file.

**See Also** `defoutput`, `open`, `quote`, `write line`

## write line (statement)

Use `write line` to write lines of data to a sequential file.

### Format

```
write line [#filenum, ] [item] [{, | ;} ...
           [item]] ... [;]
```

### Comments

The `write line` statement writes a line of data to a sequential file. This statement operates only on files opened in output or append modes.

*filenum* must be an open file output number. If *filenum* is omitted, the file number stored in the variable `defoutput` is assumed.

To write each item on a separate line, separate the items with a comma. To write the data on a single line rather than separate lines, separating the items with a semicolon.

Normally, `write line` terminates each addition to the file with a carriage-return/line-feed (CR/LF) pair. To suppress the CR/LF, use the trailing semicolon.

### Example 1

```
write line "end of test"
```

In this example, the text `end of test` is written to a file. Since the file number is not specified, the default file number in `defoutput` is used.

### Example 2

```
write line #1, some_text
```

In this example, the macro writes the contents of `some_text` to the file identified by the file number 1.

### See Also

`defoutput`, `open`, `write`

## xpos (function)

Use `xpos` to find out the column location of the cursor.

**Format**            `x = xpos`

**Comments**        `xpos` returns the number of the column in which the cursor is located.

**Example 1**        `cur_col = xpos`

In this example, the macro assigns the cursor's current column position to the `cur_col` variable.

**Example 2**        `if xpos = winsizeX - 1 then alarm`

In this example, the PC sounds an alarm if the cursor is located one column less than the size of the window.

**See Also**        `ypos`

## ypos (function)

Use `ypos` to find out the row location of the cursor.

**Format** `x = ypos`

**Comments** `ypos` returns the number of the row in which the cursor is located.

**Example 1** `cur_row = ypos`

In this example, the macro assigns the cursor's current row position to the `cur_row` variable.

**Example 2** `if ypos = winsizey - 1 then alarm`

In this example, the PC sounds an alarm if the cursor position is one row less than the size of the window.

**See Also** `xpos`

## zoom (statement)

Use `zoom` to maximize a session window.

**Format** `zoom`

**Comments** `zoom` maximizes a session window.

To maximize the Accessory Manager application window, use the `maximize` function.

**Example**

```
if online then
  zoom
```

In this example, the session window is maximized if the session is online to the host.

**See Also** `hide`, `show`, `maximize`



---

# Connection, Terminal, and File Transfer Tools

# 7

## In This Chapter

This chapter provides information on Accessory Manager's tools for connecting to a host, emulating a terminal, and transferring files. The chapter also lists the variables used by each tool.

Tool Concept .....	364
Connection Tools .....	365
Terminal Tools .....	366
File Transfer Tools .....	367
Using Tool Variables .....	368
Connection Tool Variables .....	369
T 27 Variables .....	370
UTS Variables .....	374

## Tool Concept

A tool is a code file that controls a specific aspect of a session. There are three types of tools:

- Connection tool
- Terminal tool
- File transfer tool

The tools correspond to the options on the Session Type dialog box in Accessory Manager. The connection tool corresponds to the Connection Type; the terminal tool corresponds to the Display/Device Type; and the file transfer tool corresponds to the File Transfer Protocol.

For example, a T 27 session uses the INFOConnect connection tool, the T 27 terminal tool, and the CANDE file transfer tool.

Minimally, each session must have a connection tool and a terminal tool; a file transfer tool is needed only when you want to transfer files. Each of these tools is described in detail later in this chapter.

You can configure the settings for the tools using the Settings dialog box in Accessory Manager. For example, to configure the connection tool, click Settings from the Options menu, click Connection from the Categories list box, and complete the right half of the dialog box. To configure the terminal tool, click Display from the Categories list box. To configure the file transfer tool, click File Transfer.

You can also configure many of these settings using a CASL macro. For more information, refer to “Using Tool Variables” on page 368.

## Connection Tools

The connection tool determines which mechanism the session uses to communicate with the host. For example, T 27 and UTS use the INFOConnect connection tool. This connection tool lets you select the INFOConnect path to use with your session, as well as configure other options (such as the action to take if the session is disconnected, or the host graphics protocol to use).

You can configure the INFOConnect connection tool using either the Settings dialog box in Accessory Manager or a CASL macro. For information on doing this using a CASL macro, refer to “Using Tool Variables” on page 368 and “Connection Tool Variables” on page 369.

EXTRA! Enterprise for Accessory Manager and WinFTP sessions do not use the INFOConnect connection tool; they have separate connection tools of their own.

Currently, you cannot use a CASL macro to specify which connection tool to use for EXTRA! Enterprise for Accessory Manager or WinFTP sessions. To specify the connection tool, you must click Session Type from Accessory Manager’s Options menu and click the desired item from the Connection Type list box.

In addition, you cannot use a CASL macro to configure an EXTRA! Enterprise for Accessory Manager or WinFTP connection tool. To configure the connection tool, you must click Settings from the Options menu, click Connection from the Categories list box, and complete the Settings dialog box.

## Terminal Tools

The terminal tool determines which kind of terminal the PC will emulate during a session. For example, the T 27 terminal tool emulates a T 27 terminal; the UTS terminal tool emulates a UTS 20, UTS 40, or UTS 60 terminal. Each terminal tool lets you interact with a particular type of host in the manner that the host expects.

You cannot change a session from one terminal emulation type to another. For example, you cannot change a T 27 session to a UTS session. However, you can change from one sub-model to another. For example, you can change from a UTS 20 to a UTS 60 session. For more information, refer to “[terminal \(system variable\)](#)” on page 326.

Although you cannot change the terminal tool for a session, you can configure it using either the Settings dialog box in Accessory Manager or a CASL macro. For information on doing this using a CASL macro, refer to “[Using Tool Variables](#)” on page 368, “[T 27 Variables](#)” on page 370, and “[UTS Variables](#)” on page 374.

Currently, you cannot use a CASL macro to configure the ALC or EXTRA! Enterprise for Accessory Manager terminal tools. To configure these terminal tools, you must click Settings from the Options menu, click Display from the Categories list box, and complete the Settings dialog box.

## File Transfer Tools

The file transfer tools determines which file transfer protocol to use for a session. 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.

Each product comes with its own file transfer tools. For example, T 27 comes with a CANDE file transfer tool; UTS comes with a MAPPER and OS2200 file transfer tool; 3270 comes with a IND\$FILE file transfer tool.

You cannot change to a file transfer protocol that is not supported by the session's terminal type. For example, you cannot change from CANDE to OS2200, since the former is designed for use with T 27, and the latter for use with UTS. For more information, refer to [“protocol \(system variable\)”](#) on page 288.

Currently, you cannot use a CASL macro to specify which file transfer tool to use for EXTRA! Enterprise for Accessory Manager sessions. To specify the file transfer tool, you must click Session Type from Accessory Manager's Options menu and click the desired item from the File Transfer Protocol list box.

You can configure UTS's MAPPER file transfer protocol using either the Settings dialog box in Accessory Manager or a CASL macro. For information on doing this using a CASL macro, refer to [“Using Tool Variables”](#) on page 368 and [“UTS Variables”](#) on page 374.

However, you cannot use a CASL macro to configure T 27's file transfer protocol (CANDE) or UTS's OS2200 file transfer protocol. To do this, you must click Settings from the Options menu, click File Transfer from the Categories list box, and complete the Settings dialog box.

## Using Tool Variables

The connection, terminal, and file transfer tools have predefined variables that you can read or change using a CASL macro. These variables correspond to options that you can change on the Settings dialog box. The values for these variables are stored in each session's .ADP file.

The variable names are stored in text files with file extensions of .PRE.

**Note:** Do not alter the .PRE files in any manner. Otherwise, compiling a macro becomes unpredictable.

To read or set a variable, use the `assume` statement to specify the tool type and file name. Then use the format `variable_name = value` to set the desired configuration option. For more information about the `assume` statement, refer to “[assume \(statement\)](#)” on page 134.

### Example

```
assume terminal "dcat27"  
CurShape = "block"
```

T 27 has a string variable `CurShape` that can have the values `Block`, `Underline`, or `VerticalBar`. This macro changes the cursor shape to a block.

## Connection Tool Variables

The INFOConnect connection tool supports the variables in ICSTOOL.PRE:

Variable	Type	Description	Values
DevModel	String	An internal setting that does not appear on the Settings dialog box but determines which paths appear in the Path ID list box  The DevModel must match the OpenID.	All paths Telnet/TTY paths Unassociated paths Unisys® A/V Series Paths Unisys 1100/2200 paths
DynamicPath	Boolean	If this variable is true, the user must select a path from the Select INFOConnect Path dialog box. If it is false, the path specified by PathID is used automatically.	true, false
HostGraphics	Integer	The host graphics protocol to use with the session	0=GraphX is not installed 1=GraphX for an 1100/2200 Series host 2=GraphX for an A Series host 3=GraphX for a UNIX® host
OpenID	String	An internal setting that does not appear on the Settings dialog box but determines which paths appear in the Path ID list box  To use this, UseOpenID must be set to true, and OpenID must match the DevModel.	ANSI (Telenet/TTY paths) MT (Unisys A Series paths) UTS60 (Unisys 1100/2200 paths) " " (unassociated paths)
PathID	String	The INFOConnect path to use for the session	Any valid INFOConnect path name
UseOpenID	Boolean	Determines whether the connection tool lists only the INFOConnect paths matching those specified by the OpenID	true, false

## T 27 Variables

T 27 supports the variables in DCAT27.PRE:

Variable	Type	Description	Values
AlarmLevel	Boolean	Determines whether the PC sounds a beep when the cursor reaches a specified location	true, false
AlternateRS	Integer	Character to use for the record separator field delimiter	1-255
AlternateUS	Integer	Character to use for the unit separator field delimiter	1-255
AutoSizeFont	Boolean	Determines whether the font size changes with the session window size	true, false
ClrInForms	String	The data to clear when you clear data in forms mode	Unprotected, All
ColumnAlarm	Integer	Column number at which the alarm will sound (if enabled)	1-100
Columns	Integer	Number of columns per page	1-132
CR_Interp	String	The interpretation of a received CR character	CR, CRLF
CurShape	String	Cursor shape	Block, Underline, or VerticalBar
CursorWrap	Boolean	Determines whether a word wraps to the next line	true, false
DC1_Function	String	The interpretation of a received DC1 character	LineClr, StayInRcv
DC2_Function	String	The interpretation of a received DC2 character	ToggleForms, AdvanceDCP
DelimiterVisible	Boolean	Determines whether field delimiters are displayed or replaced by blanks	true, false
DispCRSym	Boolean	Determines whether a CR entered from the keyboard is displayed	true, false
DisplayETX	Boolean	Determines whether an ETX received from the host is displayed	true, false



**Chapter 7 Connection, Terminal, and File Transfer Tools**

<b>Variable</b>	<b>Type</b>	<b>Description</b>	<b>Values</b>
DisplayRcvdCR	Boolean	Determines whether a CR received from the host is displayed	true, false
DisplayRcvdHT	Boolean	Determines whether an HT received from the host is displayed	true, false
DispTabSym	Boolean	Determines whether an HT entered from the keyboard is displayed	true, false
ETX_Advance	Boolean	Determines whether the cursor advances one position when an ETX is received	true, false
FF_Clrstabs	Boolean	Determines whether variable tabs are cleared when a form feed is received	true, false
Font	String	The name of the font to use	InterComW N, InterComW B, or a fixed-width typeface name, such as Terminal or Courier
FontSize	Integer	Point size of the font to use	Varies with the font
FormXmitToCursor	Boolean	Determines whether only the data up to the cursor be sent to the host	true, false
HostScreenInvert	Boolean	Determines whether the Host To Screen translation table will be inverted	true, false
HostScreenTable	String	File name of the Host To Screen translation table	<i>filename</i>
InsSpace	Boolean	Determines whether toggling on insert mode inserts a space at the cursor	true, false
KbdCROnly	Boolean	Determines whether the cursor stays on the current row when a CR is entered	true, false
LF_Interp	String	The interpretation of a received line feed character	LF, CRLF
LineAtATimeXmit	Boolean	Determines whether the transmit key sends only the line containing the cursor	true, false

## Chapter 7 Connection, Terminal, and File Transfer Tools

---

Variable	Type	Description	Values
LowerCase	Boolean	Determines whether lower case characters can be entered from the keyboard	true, false
NoSkipField	Boolean	Determines whether the cursor stays in the current field when the field is full or goes to the next field	true, false
Pages	Integer	Number of terminal pages	1–99
RawEightBit	Boolean	Determines whether extended characters are sent to the host	true, false
RcvModeHold	Boolean	Determines whether the PC remains in receive mode after receiving a buffer	true, false
RowAlarm	Integer	Row number at which alarm sounds (if enabled)	1–50
Rows	Integer	Number of rows per page	1–50
ScreenHostInvert	Boolean	Determines whether the Screen to Host translation table will be inverted	true, false
ScreenHostTable	String	File name of the Screen to Host translation table	<i>filename</i>
ExtendedSOSI	Boolean	Determines whether to use SO and SI to send extended characters	true, false
TranslateSOSI	Boolean	Determines whether SO and SI will be used to receive extended characters	true, false
SOH_ClrScreen	Boolean	Determines whether the start of each buffer clears the screen	true, false
SOH_ExitsForms	Boolean	Determines whether the start of each buffer exits forms mode	true, false
SpcfKeyHex	Boolean	Determines whether the Specify key sends the cursor position in hexadecimal	true, false
SpcfSendsPage	Boolean	Determines whether the Specify key sends the page number as well as the cursor position	true, false

---

**Chapter 7** *Connection, Terminal, and File Transfer Tools*

---

<b>Variable</b>	<b>Type</b>	<b>Description</b>	<b>Values</b>
SpecialScroll	Boolean	Determines whether received data causes the display to scroll	true, false
TabSize	Integer	The spacing between fixed tab stops	1-100
TabStops	String	If variable tabs are used, a string where a T represents each tab	"T T T T"
TabType	String	How tab settings are specified	Fixed, Variable
VT_PageAdvance	Boolean	Determines whether a received VT causes a page advance	true, false

## UTS Variables

UTS supports the following variables in AMUTS.PRE:

Variable	Type	Specifies	Values
AltBrightness	String	The way the cursor should blink	LowIntensity, Reverse, NormalIntensity
AlwaysHomeCursor	Boolean	Determines whether the cursor is placed at the home position even if it is protected	true, false
AutoShiftLB	Integer	The lower boundary for changing to uppercase or lowercase	1–255, indicating the character number
AutoShiftUB	Integer	The upper boundary for changing to uppercase or lowercase	1–255, indicating the character number
BeepOnSysMessage	Integer	Number of beeps when the host sends a message	0–99
BlinkEnabled	Boolean	Determines whether blink is enabled when the host sends a character with a blinking attribute	true, false
CPFlags:	Integer	Control page flags:	
		Display control characters	0x0001
		Destructive spaces	0x0002
		System response mode	0x0004
		Upper case shift	0x0008
		Keyboard click	0x0010
		Intensity of status line (UTS 20/40)	0x0100
		Ignore host color (UTS 60)	0x0200
		Sound screen alarm (UTS 60)	0x1000
		Repeat screen alarm (UTS 60)	0x2000
		Cursor return (UTS 60)	0x4000
		Sets all values to their defaults	0x3312
		<b>Note:</b> All apply to UTS 20/40/60 unless noted otherwise. For example, intensity of status line applies only to UTS 20/40.	
		To specify a combination of Control Page flags, add the values in the Value column. For example, to both display control characters and use a destructive space, set CPFlags = 0x0003.	

**Chapter 7 Connection, Terminal, and File Transfer Tools**

<b>Variable</b>	<b>Type</b>	<b>Specifies</b>	<b>Values</b>
CursorShape	String	Shape of the cursor	block, underline, verticalbar
DefaultAppName	String	Name of host application specified in Windows registry	<i>application_name</i>
DNPartialEnd	Integer	The end line of a partial file transfer in a MAPPER download	<i>line_number</i>
DNPartialStart	Integer	The start line of a partial file transfer in a MAPPER download	<i>line_number</i>
DNPCFileMode	String	The file mode for a MAPPER download	append, overwrite, insert
DNPCFileType	String	The file type for a MAPPER download	csv, textwithtabs, textnotabs
DNSilentMode	Integer	MAPPER downloads in silent mode	1=silent mode 0=off
DNStripHeader	Integer	MAPPER downloads stripping the header	1=strip header 0=off
DynamicSizing	Boolean	Determines whether the font size changes with the session window size	true, false
EmphasisTransmit	String	Type of emphasis to transmit	emphxmit_none, emphxmit_e2, emphxmit_e3
ExtendedCP	Boolean	Determines whether the extended control page is enabled	true, false
FaceName	String	Font name	UTSFONT, PEPPONT, or a fixed-width typeface name, such as Terminal or Courier
FCCTransmit	String	Type of FCCs to transmit	fccxmit_none, fccxmit_expanded, fccxmit_color
HostAutoLogon	Boolean	Determines whether automatic logon occurs when the session is started	true, false
HSTableName	String	Name of Host To Screen translation table	<i>filename</i>
OverrideHostFCCs	Boolean	Determines whether host FCC changes are overridden	true, false

## Chapter 7 Connection, Terminal, and File Transfer Tools

Variable	Type	Specifies	Values
Pages	Integer	Number of pages	1–9
PointSize	Integer	Size of the font	Varies with the font
PrintArea	String	Specifies which data to print	<code>prange_soecursor</code> , <code>prange_fullpage</code> , <code>prange_selected</code>
PrinterDID	Integer	Device identifier where the host should send host-initiated print jobs	A valid DID value (hexadecimal)
PrintMode	String	Controls the way data on the screen is printed	<code>print_form</code> , <code>print_prnt</code> , <code>print_xpar</code>
ProtCPPageColor	Integer	Color of protected characters in the control page	<i>BgFg</i> (hexadecimal)
ReadDID	Integer	Device identifier that will receive data from a device such as a host disk drive or tape system	A valid DID value
SaveHostCPChanges	Boolean	Determines whether to save any Control Page settings sent by the host	<code>true</code> , <code>false</code>
ScanBackOn ProtectedFields	Boolean	Determines whether the cursor goes to the previous unprotected character when you try to put cursor on a protected field using the arrow key	<code>true</code> , <code>false</code>
ScreenColor	Integer	Color of screen	<i>BgFg</i> (hexadecimal)
SHTableName	String	Name of Screen To Host translation table	<i>filename</i>
SpLEOLProcessing	Boolean	Determines whether the PC scans for an end-of-line or end-of-field character	<code>true</code> , <code>false</code>
StatusLineColor	Integer	Color of status bar	<i>BgFg</i> (hexadecimal)
TerminalType	String	The terminal type	UTS20, UTS40, UTS60
TransmitMode	String	Controls how data is transmitted	<code>xmit_all</code> , <code>xmit_chan</code> , <code>xmit_var</code>
UnprotCPPageColor	Integer	Color of unprotected characters in the control page	<i>BgFg</i> (hexadecimal)

---

**Chapter 7** Connection, Terminal, and File Transfer Tools

---

<b>Variable</b>	<b>Type</b>	<b>Specifies</b>	<b>Values</b>
UPInsertLine	String	Indicates the line in the MAPPER report where the insertion should begin	<i>line_number</i>
UPMapperCommand	Integer	MAPPER command character	<i>character</i>
UPMaxLines	Integer	The number of lines downloaded at a time in a MAPPER upload	<i>number</i>
UPPartialEnd	Integer	The end line of a partial file transfer in a MAPPER upload	<i>line_number</i>
UPPartialStart	Integer	The start line of a partial file transfer in a MAPPER upload	<i>line_number</i>
UPPCFileMode	String	The action to take if data already exists in the MAPPER report	append, overwrite, insert
UPPCFileType	String	The file type for a MAPPER upload	csv, textwithtabs, textnotabs
UPSilentMode	Integer	MAPPER upload in silent mode	1=silent mode 0=off
WSCols	Integer	Number of columns per page	2-132
WSFCCs	Integer	Maximum number of FCCs per page	<i>number</i>
WSRows	Integer	Number of rows per page	2-50





---

# *Error Messages*

# A

## **In This Appendix**

This appendix includes the following headings:

Classes of Error Message .....	380
Internal Errors .....	381
Compiler Errors .....	382
Input/Output Errors .....	390
Mathematical and Range Errors .....	392
State Errors .....	393
Critical Errors .....	394
Macro Execution Errors .....	395
Compatibility Errors .....	398
Upload/Download Errors .....	399
Missing Information Errors .....	400
Multiple Document Interface Errors .....	401
Emulator or File Transfer Protocol Errors .....	402
DLL Errors .....	403
Generic Module Errors .....	404
File Transfer Errors .....	405
Navigation Errors .....	407

## Classes of Error Message

The tables on the following pages list the error messages that might appear while you are compiling or running CASL macros, as well as possible solutions to these problems.

The following table lists error message classes and a description of each class. A class number precedes each error number.

<b>Class</b>	<b>Description</b>
10	Internal errors
12	Compiler errors
13	Input/output errors
14	Mathematical and range errors
15	State errors
16	Critical errors
17	Macro execution errors
18	Compatibility errors
19	Upload/download errors
21	Missing information errors
23	Multiple Document Interface errors
28	Emulator or file transfer protocol error
33	DLL errors
40	Generic module errors
45	File transfer errors
50	Navigation errors

## Internal Errors

Error Code	Error Message	Explanation
10-08	Internal error: Cannot find a connection, file transfer, or terminal tool. All tools must be installed to the frame directory before running Accessory Manager.	<p>When you run Accessory Manager, it refers to the GI32.INI file for a list of installed connection, terminal, and file transfer protocols. There must be at least one of each. This error can occur under the following circumstances:</p> <ul style="list-style-type: none"> <li>▪ No terminal emulator has been installed. Install a terminal emulator (such as UTS or T27) before running Accessory Manager.</li> <li>▪ The GI32.INI file has been moved or deleted. Put a copy of the GI32.INI file in your Windows directory, or reinstall Accessory Manager.</li> <li>▪ The GI32.INI file has been modified, and Accessory Manager cannot read it. Delete the GI32.INI file and reinstall Accessory Manager.</li> </ul>
10-12	Internal error: Unknown GI error.	An internal error has occurred. Contact Customer Support.
10-49	Internal error: Bad row number.	Your CASL macro has set an invalid row number. Edit the macro to ensure that the row number is valid.
10-50	Internal error: Bad column number.	Your CASL macro has set an invalid column number. Edit the macro to ensure that the column number is valid.
10-51	Internal error: Bad length.	The length of data in your CASL macro is invalid. Edit the macro to ensure that the data length is valid.
10-96	Unrecognized error code.	An internal error has occurred. Contact Customer Support.

## Compiler Errors

Error Code	Error Message	Explanation
12-001	Too few arguments to <i>procedure/ function</i> ' <i>procedure/ function name</i> '.	When calling a previously defined function or procedure, you specified more arguments than you originally defined. Check the definition of the referenced procedure or function, and correct your macro.
12-002	Too many arguments to <i>procedure/ function</i> ' <i>procedure/ function name</i> '.	When calling a previously defined function or procedure, you did not specify all the arguments that you originally defined. Check the definition of the referenced procedure or function, and correct your macro.
12-003	Array ' <i>array name</i> ' is too large.	Arrays are limited to a size of 32 KB. The referenced array exceeds that size. You can calculate the size of an array by multiplying the size of the data elements by the total number of elements in the array. Redefine the size of your array.
12-004	Invalid left hand side of assignment statement.	The operand on the left side of the assignment statement is invalid and cannot be assigned a value. This operand must be a variable. You cannot assign a value to a procedure, function, or constant. Correct the assignment statement and try again.
12-005	Bad combination of type modifiers.	The modifiers of this declaration are mutually exclusive. Modify the statement and try again.
12-006	No more cases allowed after the default case.	The default case must be the last value in a case statement. Check the structure of the case statement.
12-007	This format of the <i>statement name</i> statement is not supported in this version.	The statement in the macro is not supported or is incorrectly formatted. Refer to <a href="#">Chapter , "CASL Language,"</a> for the correct syntax.
12-008	End of file was encountered in a comment.	The compiler reached the end of the source file while processing a comment. Check to see if the end-of-comment delimiter was accidentally deleted.
12-009	<i>language element</i> must be a compile time constant.	You must use a constant. You cannot use a variable.
12-018	Duplicate declaration of ' <i>variable</i> '.	You have declared this variable twice. Only one declaration is allowed.

## Appendix A Error Messages

Error Code	Error Message	Explanation
12-019	Reference to undeclared variable ' <i>variable</i> '.	This variable has not been declared, and the compiler was unable to determine its data type from the context. Declare the variable in your macro.
12-020	Division by zero.	In evaluating the expression in this statement, you attempted to divide by zero. This is not allowed. Correct your macro and try again.
12-021	Unable to open file ' <i>bad file</i> '.	The compiler received an error when it tried to open this file. Check that the file name is specified correctly.
12-022	Error reading file ' <i>bad file</i> '.	The compiler encountered an error while trying to read this file. Make sure the file exists and is not damaged.
12-023	For loop needs assignment.	You did not set the initial value of the loop control variable in a for statement. Correct the for statement in your macro.
12-024	' <i>procedure/ function name</i> ' was declared forward as <i>procedure</i> or <i>function</i> , not <i>procedure</i> or <i>function</i> .	One of two things occurred: <ul style="list-style-type: none"><li>■ You declared this procedure or function as a procedure in the forward declaration, but defined it as a function in the actual definition.</li><li>■ You declared this procedure or function as a function in the forward declaration, but defined it as a procedure in the actual definition.</li></ul> Correct your macro so the forward declaration and the definition match.
12-025	Too few parameters to ' <i>procedure/ function name</i> ' to match forward declaration.	The definition of this procedure or function has fewer parameters than its forward declaration. Make sure the forward declaration and the actual definition match exactly.
12-026	Too many parameters to ' <i>procedure/ function name</i> ' to match forward declaration.	The definition of this procedure or function has more parameters than its forward declaration. Make sure the forward declaration and the actual definition match exactly.
12-027	Unresolved forward <i>procedure</i> or <i>function</i> ' <i>procedure/ function name</i> '.	You made a forward declaration for this procedure or function, but you never provided an actual definition of it. Provide a definition for this procedure or function in your macro.
12-028	' <i>identifier</i> ' is not a function name.	You have used an identifier as a function, but it is not a function. You must use a valid function name.

## Appendix A Error Messages

---

Error Code	Error Message	Explanation
12-029	genlabels directive must be on to use a computed goto.	At some point in your macro, you specified <code>genlabels off</code> . This directive must be on (its default state) to use the <code>goto</code> statement in a macro.
12-030	' <i>identifier</i> ' is not a label.	You have used <code>identifier</code> as a label, but it is not a label. You must use a valid label.
12-031	Input statement needs a variable, not a constant.	You must specify a variable rather than a constant for the input statement. The input statement will use this variable to process keyboard input.
12-032	Internal error: <i>compiler module line number</i> .	An internal error has occurred in the compiler. Contact Customer Support and be prepared to furnish a copy of the macro that caused the error along with the exact information in this message.
12-033	Invalid time interval.	You specified a time interval incorrectly. Check the way you expressed the time.
12-034	Unresolved label: ' <i>identifier</i> '.	This label was never defined anywhere in your macro. Add the label to the appropriate section of your macro.
12-035	Lexical analysis error: <i>specific error</i> .	This error occurred during the lexical analysis phase of the compilation process. Check this section of your macro for syntax errors.
12-036	List box contents must be <i>string</i> or one-dimensional <i>string</i> array.	The variable that contains the list of items to be included in a list box must be either a string of items separated by commas or a one-dimensional array of strings.
12-037	Compiler out of memory.	The compiler ran out of memory while compiling your macro. Close any unneeded applications and try again.
12-038	Too many arguments to <code>Nextline</code> .	Too many arguments were specified for the <code>nextline</code> statement. Check the list of arguments you are passing to this statement.
12-040	Second operand of <code>mod</code> operator must be positive.	The modulus function allows only positive numbers for its second operand. Revise your statement to use a positive number.
12-042	Cannot have more than one OK or Cancel button.	A dialog box can have only one OK button and one Cancel button. Revise your macro accordingly.
12-043	Could not open module file ' <i>bad file</i> '.	Accessory Manager could not open the module file you specified. Make sure that the file name is correct and that the file resides in the proper location.

---

## Appendix A Error Messages

---

Error Code	Error Message	Explanation
12-044	Parsing error: <i>specific error</i> .	This error occurred during the syntactic analysis phase of the compilation process. Check this section of your macro for syntax errors.
12-045	Print format specification is not supported in this version.	Accessory Manager does not support print format specifications. Revise your macro to eliminate these specifications.
12-046	' <i>identifier</i> ' is not a procedure name.	You have used an identifier as a procedure, but it is not a procedure. You must use a valid procedure name.
12-047	Exit can only be used inside a procedure.	The compiler encountered an <code>exit</code> statement outside of a procedure. Check the procedures and functions in your macro and make sure that they begin and end properly.
12-048	Return with value can only be used inside a function.	The compiler encountered a return with a value outside of a function. Values can only be returned from functions. Check the procedure and functions in your macro begin and make sure that they begin and end properly.
12-049	Exit cannot be used in a function.	The <code>exit</code> statement cannot be used to leave a function. It can only be used to leave procedures. Use the <code>return</code> statement instead of the <code>exit</code> statement in a function.
12-050	Return in a procedure cannot return a value.	Procedures cannot return values. The <code>return</code> statement is used to return a value inside a function. Either redefine your procedure as a function, or change the <code>return</code> statement in your procedure.
12-051	Bad use of '^' in string constant.	The caret symbol followed by a control character indicates an unprintable control character in a string constant. The character following the caret is not a valid control character. Check the character following the caret in the string constant.
12-052	String constant too long.	The maximum length of a constant is 256 characters. Shorten your string to fit within this limit.
12-053	String subscript out of range.	The subscript you specified to access a character in this string is beyond the end of the string. Make sure the subscript is within the bounds of the string.
12-054	Too few subscripts to array name.	You have not specified enough subscripts to reference this array. You specified more dimensions when you declared the array than you used when you referenced it. Correct either the declaration or the reference.

## Appendix A Error Messages

---

Error Code	Error Message	Explanation
12-055	Too many subscripts to array name.	You have specified too many subscripts to reference this array. You specified fewer dimensions when you declared the array than you used when you referenced it. Correct either the declaration or the reference.
12-056	Syntax error at 'bad token'.	The compiler found an error in your macro near <i>bad token</i> . Make sure that all language elements in this section of your macro are specified properly.
12-057	Bad token: 'string'.	The compiler did not recognize a string in your macro. Make sure that all language elements in string, and in the instructions surrounding it, are specified properly.
12-058	Track procedure cannot take parameters.	The procedure you named to the <code>track</code> statement cannot have any parameters. Make sure that both the <code>track</code> statement and the procedure definition are specified properly.
12-059	Track procedure cannot be a function.	The procedure you named to the <code>track</code> statement must be a procedure, not a function. Make sure that both the <code>track</code> statement and the procedure definition are specified properly.
12-060	Track procedure can only be a label or user procedure.	The procedure you named to the <code>track</code> statement may only be a procedure or a label. Make sure that both the <code>track</code> statement and the procedure definition are specified properly.
12-061	Type error: Assume file name must be a string constant.	The file name you specified in the <code>assume</code> statement must be a string constant, not a variable. Make sure that the name is a string and a constant.
12-062	Type error: cannot perform "operator" on types type 1 and type 2.	This operation cannot be performed on variables of these types. Check the operation and make sure that the operands are of compatible types.
12-063	Type error: case selector cannot be bad type.	The type specified in the message cannot be used for the selector in a <code>case</code> statement. Use a different type for the selector.
12-064	Type error: cannot convert type 1 to type 2.	The compiler cannot convert the values specified. Check the operation and make sure that the operands are of compatible types.



---

**Appendix A Error Messages**

---

<b>Error Code</b>	<b>Error Message</b>	<b>Explanation</b>
12-065	Type error: " <i>string</i> " cannot be converted to <i>type</i> .	The compile encountered an error when attempting to convert this string into <i>type</i> . This conversion was required by the usage of the string in your macro. Make sure the value in this string is compatible with the data types required by this statement. Perhaps a string is not required in this case and some other data type could be used.
12-066	Type error: <i>language element</i> must be <i>good type</i> , not <i>bad type</i> .	You used an invalid type for <i>language element</i> . You must use the type specified in <i>good type</i> .
12-067	Type error: <i>language element</i> must be a <i>type</i> variable.	You used an invalid type for <i>language element</i> . You must use a variable of the type specified in <i>type</i> . A constant is not allowed in this situation.
12-068	Type error: Parameter <i>number</i> of ' <i>procedure/function name</i> ' was declared forward as <i>good type</i> , not <i>bad type</i> .	In the forward declaration of this procedure or function, this parameter was declared to be of a different type than in the actual definition. Make sure the forward declaration and the actual definition match exactly.
12-069	Type error: Return type of ' <i>function name</i> ' was declared forward as <i>good type</i> , not <i>bad type</i> .	In the forward declaration of this function, the return value was declared to be of a different type than in the actual definition. Make sure the forward declaration and the actual definition match exactly.
12-070	Type error: colors must be <i>integer</i> or specific color names.	You must either use an integer expression or specific color names, such as "red," to specify a color.
12-071	Type error: argument <i>number</i> of <i>procedure</i> or <i>function</i> ' <i>procedure/function name</i> ' must be <i>good type</i> , not <i>bad type</i> .	One of the arguments for this procedure or function is of the wrong type. Check the definition of the procedure or function and make sure that you are calling it properly.
12-072	Type error: cannot subscript <i>variable</i> .	This variable is not an array variable and cannot be subscripted. Either declare the variable to be an array, or use an existing array variable.
12-073	Type error: subscript ' <i>number</i> ' of ' <i>array name</i> ' must be <i>good type</i> , not <i>bad type</i> .	This subscript is of the wrong type. Make sure the subscript is of the type specified in <i>good type</i> .

---

## Appendix A Error Messages

---

Error Code	Error Message	Explanation
12-074	Type error: subscript ' <i>string name</i> ' must be <i>good type</i> , not <i>bad type</i> .	This subscript is the wrong type. Make sure the subscript is of type specified in <i>good type</i> .
12-075	Type error: cannot perform ' <i>operator</i> ' on type <i>bad type</i> .	This operation cannot be performed on a variable of <i>bad type</i> . Check the operation and make sure that the operand's type is compatible with the operation.
12-076	Type error: <i>procedure</i> must be a user procedure.	A user-defined procedure is required here. You cannot use a CASL built-in procedure.
12-077	The number of buttons is limited to four.	An alert box can have only four buttons. You have tried to put too many buttons in your box. Limit the number of buttons to four.
12-078	Statement or expression is too complex.	This statement or expression is too complex for the compiler to compile. Simplify the statement or expression, or break it up into smaller parts.
12-079	Type error: cannot assign <i>right-side type</i> to <i>left-side type</i> .	The type of expression on the right side of the assignment statement is not compatible with the variable on the left side. Correct the assignment statement to make the types agree.
12-080	Error writing file ' <i>bad file</i> '.	The compiler received an error from the file system when it tried to write to the specified file. Possible reasons for this error are as follows: <ul style="list-style-type: none"><li>▪ Your disk is full. Free up space on this disk or use another disk.</li><li>▪ You have too many files open in other applications. Close any applications you are not using.</li><li>▪ Your disk is bad. Check to make sure your disk is not damaged.</li><li>▪ A removable disk or a network disk is no longer online. Make sure the disk you are trying to write to is online.</li></ul>
12-081	String constant must be one line.	A string constant must be entirely on one line. It cannot extend across multiple lines. Your string is too long. Make sure the string has a closing quotation mark.

---

## Appendix A Error Messages

---

Error Code	Error Message	Explanation
12-082	Keyword ' <i>bad-keyword</i> ' cannot be used here.	The referenced CASL keyword cannot be used in this context. If you were not aware that this was a CASL keyword, you can correct this problem by adding <i>the</i> or <i>my</i> to the word. For example, you can use <i>my_password</i> rather than <i>password</i> .
12-255	Unrecognized keyword: ' <i>bad keyword</i> ' .	The keyword is not known by the compiler. Revise your macro to eliminate this keyword.

## Input/Output Errors

Error Code	Error Message	Explanation
13-05	The file number is invalid or missing.	Make sure you specify a file number in the <code>get</code> , <code>put</code> , <code>read</code> , and <code>write</code> statements. You must precede the number with the pound symbol (#).
13-06	The specified file channel number is already open. You must first close the channel or use another one.	The specified file channel number is already open. You must first close the channel or use another one.
13-07	The specified channel number is not open.	You tried to manipulate a file using <code>read</code> , <code>write</code> , <code>get</code> , or <code>put</code> without first opening the file, or the file was previously closed. Open the file before using <code>read</code> , <code>write</code> , <code>get</code> , or <code>put</code> .
13-08	Accessory Manager cannot read an output file.	You opened this file for output only and tried to issue a <code>read</code> or <code>get</code> statement. Modify your macro and try again.
13-09	Accessory Manager cannot write to an input file.	You opened this file for input only and tried to write to it using the <code>write</code> or <code>put</code> statements. Modify your macro and try again.
13-10	Accessory Manager cannot get/put a text file.	You opened the file for input or output. These are read and written to sequentially using the <code>read</code> and <code>write</code> statements. Use <code>get</code> and <code>put</code> for random files.
13-11	Accessory Manager cannot read from or write to a random file.	You opened the file in random mode and tried to use the <code>read</code> or <code>write</code> statements. Use <code>get</code> and <code>put</code> for random files.
13-16	Window coordinates out of range.	The coordinates you have specified for accessing a window are not valid. The coordinates must access a valid portion of the window or display.
13-18	The specified window is not open.	You have specified a window that is not open. You cannot perform operations on a window unless it is open.
13-28	Attempt to send output to the display failed.	An error occurred while Accessory Manager was trying to write information to the screen. Try running the macro again. If it still fails, exit Accessory Manager and/or Windows and try again.

---

## Appendix A Error Messages

---

Error Code	Error Message	Explanation
13-29	A file copy failed.	Accessory Manager was unable to copy a file. The following are possible reasons for this error: <ul style="list-style-type: none"><li>▪ Your disk is full. Delete unneeded files and try again.</li><li>▪ You have too many files open in other applications. Close the open files and try again.</li><li>▪ Your disk is bad. Contact your system administrator.</li><li>▪ A removable disk or a network disk is no longer online. Try again when the specified disk is online.</li></ul>
13-30	The script attempted a seek in a sequential file; you can use seek only with random files.	The file was not opened properly for performing the seek function. Open the file using the appropriate mode.
13-31	Multiple windows in a session are not supported in this version.	This feature is not currently supported. Revise your macro to use other language elements.
13-32	An error has occurred in attempting to create a new window.	An error occurred with the new command in your macro. If you are using this command to open an existing session, be sure to specify the file name of the existing session.
13-33	There is already a file that has the name selected.	You must use a unique name for each file. Change the file name and try again.
13-48	File creation error.	Accessory Manager was unable to create a file. Verify that you have adequate space on your disk and that you have write privileges.
13-64	You must use -k or -c when using -p command line parameter.	The -p command line parameter specifies which INFOConnect path to use for a particular session. You must first open a session using the -k or -c command line parameters before you can specify a path for the session.
13-65	The caption specified is too long. It will be truncated.	The caption specified for the session window title bar is greater than 128 characters. Accessory Manager will truncate the caption unless you reduce its size.
13-66	Administration utility file not found. See your administrator for further instructions.	If the file AMFULL.RCF is not in the Windows directory, Accessory Manager cannot run. Copy this file to the Windows directory, or reinstall Accessory Manager.

## Mathematical and Range Errors

Error Code	Error Message	Explanation
14-03	Division by zero was attempted.	You tried to divide by 0. Check your macro, and the expression used for the divisor, to determine why the divisor contained a value of 0.
14-05	The expression is not valid for the variable.	You tried to assign a different variable type to this variable. Be sure to use valid expressions for each variable.
14-06	The value is outside the permissible range.	You specified a range for the indexes in an array variable. The index falls outside that range.
14-09	A string was truncated.	Accessory Manager truncated a string because it was too long. Strings can be up to 32 KB.
14-10	Invalid characters were found in a numeric string.	You tried to make an assignment to an integer value. The expression contained alphabetic or non-numeric characters. If you are using hexadecimal representation, make sure the number ends in <i>h</i> .
14-11	The specified value is outside the acceptable range.	You specified a range for the indexes in an array variable. The index falls outside that range. Increase the size of the array. If you are using a variable for the index, make sure that the variable contains a value inside the defined array range.
14-18	An invalid string was specified for the quote function.	A string specified for the <code>quote</code> function cannot contain both single and double quotation marks. Make sure that both types of marks are not used in the string you pass to the <code>quote</code> function.

## State Errors

<b>Error Code</b>	<b>Error Message</b>	<b>Explanation</b>
15-01	The specified command is applicable only when you are online.	You were running a macro meant to be used online, and you were not connected to a host. You may want to use the <code>trap</code> , <code>error</code> , and <code>online</code> functions in the macro to determine if you are connected.
15-07	The specified session does not currently exist.	This function requires a session number as a parameter. Make sure the session exists by using the <code>sessno</code> function to get its session number.
15-08	Feature is not supported by the current terminal.	Modify your macro to ensure that only valid functions for the specified terminal type are executed.

## Critical Errors

<b>Error Code</b>	<b>Error Message</b>	<b>Explanation</b>
16-02	Drive is invalid or unknown.	Specify a valid drive and try again.
16-03	Drive is not ready.	Insert a disk or close the drive door.
16-07	A seek error has occurred.	Accessory Manager could not find the specified data. Use the CHKDSK utility to make sure your disk has not been corrupted.
16-11	A write fault has occurred.	Accessory Manager could not find the specified data. Use the CHKDSK utility to make sure your disk has not been corrupted.
16-12	A read fault has occurred.	Accessory Manager could not find the specified data. Use the CHKDSK utility to make sure your disk has not been corrupted.



## Macro Execution Errors

Error Code	Error Message	Explanation
17-01	The specified label cannot be found.	Make sure the label you specified in the <code>gosub</code> or <code>goto</code> statements has a corresponding label statement where you want it to go. Labels are not case-sensitive.
17-03	'gosub' statements are nested too deep.	You can have only a certain number of <code>gosub</code> statements without issuing a return. Refer to <a href="#">Chapter , "CASL Language,"</a> for the correct syntax.
17-05	A data type mismatch for an external variable was found.	You are referencing a variable declared in another macro. Check the other macro for the appropriate data type for that variable.
17-07	The script was canceled by the user.	This is an informational message. You can run the macro again.
17-08	A reference to an unresolved external variable was found.	This variable is declared as external in this macro. It must be declared as public in a macro that calls this macro using the <code>do</code> statement.
17-10	An unavailable module variable was found.	The module in the <code>assume</code> statement is not yet loaded. Use the <code>device</code> , <code>terminal</code> , or <code>protocol</code> system variables to load a given tool. The <code>assume</code> statement only makes these variables and settings known to the compiler; it does not load the tool to make it accessible to running macros.
17-12	A 'return' statement without a corresponding 'gosub' statement was found.	While executing the macro, a <code>return</code> statement was encountered, but the macro is not in a <code>gosub</code> . There may be a logic error in the macro. Examine the logic of the macro carefully and revise it.
17-14	A script compilation failed when 'chain', 'do', or 'compile' statement was executed.	When a <code>chain</code> , <code>do</code> , or <code>compile</code> statement is issued, Accessory Manager checks to see if the macro needs compiling. If it does, Accessory Manager recompiles it before it runs. This error message appears when a macro is compiled in this manner, but has an error and cannot continue. Use the CASL Macro Editor to correct errors in the macro, and try again.
17-15	A return value was missing in the return from a function.	You declared a function, but never used the <code>return</code> statement to return a value. The value must be the same data type you used when you declared the function.

## Appendix A Error Messages

---

<b>Error Code</b>	<b>Error Message</b>	<b>Explanation</b>
17-16	Generic error.	This error can occur when the PC is out of memory. Close any unneeded applications, and try again.
17-17	An internal error occurred. Delete the .xwc file and recompile the script.	The .XWC file has become corrupted. Delete the file and recompile the macro.
17-18	An invalid count expression was used.	The count expression used in this statement is not valid. Correct this portion of the statement.
17-19	A string expression is too long.	Strings are limited to 32 KB in size. Change the logic of your macro so that you do not create strings exceeding this length.
17-20	There is not enough global memory available.	Accessory Manager does not have enough memory to perform the function. Try closing sessions, QuickPads, and other windows that you are not currently using.
17-21	A 'dialogbox' keyword was used outside a 'dialogbox' statement.	The keywords which describe a dialog box can only be used inside a dialogbox statement. Revise your macro to eliminate this occurrence of the keyword.
17-22	'dialogbox' statements are nested. These statements cannot be nested.	Revise your macro to eliminate nested dialogbox statements.
17-23	The dialog cannot be displayed.	The dialog is too complex to be displayed. Simplify the dialog box or break it into multiple dialog boxes.
17-24	No pushbutton was specified for a dialog box.	Every dialog box must have at least one button so that the user can close the dialog box. Add at least one button to your dialog box.
17-25	'watch' statements cannot be nested.	Revise your macro so that a second watch statement is not called while another watch is active.
17-26	Too many track channels are open.	Check your usage of the track statement and reduce the number of channels being used at once.
17-27	A stack overflow has occurred. Procedures or functions are nested too deep.	You have made too many nested calls to procedures and functions. Revise your macro so that calls are not nested as deeply.

---

**Appendix A Error Messages**

---

<b>Error Code</b>	<b>Error Message</b>	<b>Explanation</b>
17-28	The specified QuickPad file cannot be found.	Make sure that you have specified the correct drive, directory, file name, and file extension for the QuickPad. If you are trying to access the QuickPad file from a network drive, make sure that you are still connected to the network.
17-29	The specified QuickPad file has not been loaded.	You have referred to a QuickPad file that is not loaded. Load the QuickPad file and then perform other operations on it.
17-30	Cannot compile script because the compiler is already compiling another script.	You can compile only one macro at a time. Wait for the first compilation to finish before starting another.

---

## Compatibility Errors

<b>Error Code</b>	<b>Error Message</b>	<b>Explanation</b>
18-01	One or more specified modules are of an incompatible version.	Your GI.DLL file is incompatible with Accessory Manager. Reinstall Accessory Manager.
18-03	The .XWC file is bad. Recompile the .XWS file.	You must recompile the .XWS file.
18-05	The specified feature is not supported in this version.	Modify your macro to ensure that only valid functions for the specified terminal type are executed.
18-16	Invalid profile.	A problem has been detected in your file. Create a new file and try again.
18-17	Section not found in profile.	A problem has been detected in your file. Create a new file and try again.
18-19	Keyword not found in profile.	A problem has been detected in your file. Create a new file and try again.
18-20	Invalid keyword in settings.	A problem has been detected in your file. Create a new file and try again.
18-21	Invalid value in settings	A problem has been detected in your file. Create a new file and try again.
18-22	Profile section read error.	A problem has been detected in your file. Create a new file and try again.

## Upload/Download Errors

Error Code	Error Message	Explanation
19-01	An unexpected DOS error has occurred.	An unexpected error occurred. Contact Customer Support.
19-02	The specified file cannot be found.	Verify that the specified drive, directory, and file name are correct.
19-03	The specified path cannot be found.	Verify that the specified drive and directory are correct.
19-05	Access has been denied to the specified file.	You do not have access privileges to the specified file, or the file is write-protected. Make sure the attributes for the file are not read-only and that the disk is not write-protected.
19-13	An invalid file name was specified.	The file name is not valid. Correct the file name and try again.
19-14	Nonexistent file specified.	The specified file name does not exist. Type a valid file name and try again.
19-15	Nonexistent directory specified.	The specified directory name does not exist. Type a valid directory name and try again.
19-19	Diskette is write-protected.	You cannot write to the specified disk. Use a different disk, or obtain write privileges.
19-21	Disk full.	The disk is full. Delete unneeded files from the disk and try again.
19-22	Invalid filename.	The specified file name is not valid. Type a valid file name and try again.
19-23	Invalid directory name.	The specified directory name is not valid. Type a valid directory name and try again.
19-24	Cannot run application specified.	The specified application cannot be run. Make sure that the application name is specified properly or try another application.

## Missing Information Errors

Error Code	Error Message	Explanation
21-01	The specified script file cannot be found. Check the name and make sure the file is in the ACCMGR directory.	Accessory Manager cannot find the specified macro file. Check the name, make sure the file is in Accessory Manager directory, and try again.
21-09	There is no default file name; 'filefind' must be used to set up a default file.	The first time that you call filefind you must specify a legal file specification that can include drive specifiers and directory names as well as wildcard characters. Only on subsequent calls can you omit the string to receive additional file names in the list.
21-10	The ADP file contains a reference to an unknown tool.	The session profile is using a connection type, terminal type, or file transfer protocol that Accessory Manager no longer recognizes. Open the session and reconfigure it using valid tools, or edit the .ADP file using a text editor.

## **Multiple Document Interface Errors**

<b>Error Code</b>	<b>Error Message</b>	<b>Explanation</b>
23-08	Unable to create an MDI document window. Try freeing some memory.	Before trying this operation again, close other open applications.

---

## Emulator or File Transfer Protocol Errors

Error Code	Error Message	Explanation
28-16	Invalid module or module not found.	A connection, terminal type, or file transfer protocol specified in your session profile cannot be found. Make sure the tools have been installed. If this error persists, recreate the session.

---



## DLL Errors

Error Code	Error Message	Explanation
33-01	DLL file could not be found.	Accessory Manager could not find a required DLL file. Verify that all the files are in Accessory Manager directory.
33-02	Path for DLL was not valid.	The directory specified for a required DLL file does not exist. Verify that all the files are in Accessory Manager directory.
33-03	DLL file was invalid or corrupt.	Reinstall Accessory Manager to overwrite the corrupt DLL file.
33-04	Unable to use requested DLL file.	Accessory Manager could not access a required DLL file. Make sure that you have read privileges to Accessory Manager directory and try again.
33-05	Unable to use requested DLL function.	Accessory Manager could not access a required DLL function. Make sure that you have read privileges to Accessory Manager directory and try again.
33-06	Attempt to use a data type that is not supported.	Refer to <a href="#">Chapter 2, "Understanding the Basics of CASL,"</a> for information about the types of data supported.

## Generic Module Errors

Error Code	Error Message	Explanation
40-16	Invalid module or module not found.	You tried to open a session that uses a terminal type that has not yet been installed or is not listed in the GI32.INI file. Use a different session, or install the desired terminal emulator, or modify the GI32.INI file to indicate that the terminal emulator has been installed.
40-17	[No error message]	No printer is currently associated with this session. Click Print Screen from Accessory Manager File menu and select a printer.
40-18	Could not locate and load library.	Accessory Manager cannot find the error strings .DLL (DCAAMERR.DLL). Reinstall Accessory Manager and try again.

## File Transfer Errors

Error Code	Error Message	Explanation
45-01	General time-out.	A general time-out has occurred. The host's protocol did not respond. Try increasing the timing specified for your file transfer protocol.
45-02	Host not responding.	The host is not responding. Accessory Manager tried to transfer the file, but received no response from the host. Check the communications link and try the transfer again.
45-04	Too many errors - transfer canceled.	Accessory Manager automatically canceled the transfer because the maximum number of errors was reached. Try again. If the problem persists, change the timing for the file transfer protocol or raise the number of errors that are allowed.
45-06	Sequencing failure - transfer canceled.	Accessory Manager canceled the transfer because of a sequencing failure. The file transfer protocol encountered an internal error. Try the transfer again. If the problem persists, contact Customer Support.
45-07	Transfer canceled by local operator.	The user canceled the file transfer. This is an informational message only. You can transfer the file again.
45-08	Transfer canceled by host.	The host canceled the file transfer. Too many errors may have occurred, or the host disk may be full. Check the host disk or increase the maximum number of errors allowed.
45-09	Protocol can't do wildcard transfers.	You used a file transfer protocol that does not support a wildcard transfer for the file name. Transfer a single file at a time or use a protocol that allows wildcard transfers.
45-11	Local disk full.	The file transfer cannot take place or was canceled because the local disk is full. Clear some space on the specified disk drive or change drives.
45-12	Host disk full.	The file transfer did not occur because the host disk is full. Clear some space on the specified host drive or change drives.

## Appendix A Error Messages

---

<b>Error Code</b>	<b>Error Message</b>	<b>Explanation</b>
45-16	Bad protocol selection.	Accessory Manager does not support the file transfer protocol you selected. Choose a supported protocol and try again.
45-18	The server command is not valid.	You issued a Kermit command that is not currently supported. Revise your macro to remove this command.

---

## Navigation Errors

Error Code	Error Message	Explanation
50-176	Error in navigation. An attempt to follow a path took us to an unknown screen. Playback is terminated.	While using the recorded navigation paths, Accessory Manager got to a screen that it could not identify. This can occur if the original recording included data that does not always appear on the host screen, or that has changed since the original recording was made. You might have to delete or truncate an identification field and try again. Refer to the online Help for information on this procedure.
50-177	Error in navigation. An attempt to follow a path took us back to the same screen. Playback is terminated.	You recorded a procedure that invokes the same host screen, or Accessory Manager cannot distinguish between two very similar host screens. Re-record the host screens and try again, or modify the identification fields for the recorded screens and try again.
50-178	Error in navigation. An attempt to follow a path took us to an unexpected screen. Playback is terminated.	While using the recorded navigation paths, Accessory Manager went to a screen that could be identified, but this was not the screen it expected to arrive at as a result of following the navigation path. Re-record the procedure to arrive at the desired host screen and try again.
50-182	No path exists from the current screen to the destination screen.	You clicked the name of a recorded host screen on the Bookmarks/Pages dialog box, but no navigation path exists to get to that screen. Re-record the procedure to get from the current screen to the desired screen and try again.



---

# Index

3270 sessions (see EXTRA! Enterprise for Accessory Manager)  
5250 sessions (see EXTRA! Enterprise for Accessory Manager)

## A

Abbreviations, used in this guide **xx**  
abs function **117**  
activate statement **118**  
activatesession statement **119**  
Addition operator **47**  
alarm statement **120**  
ALC **2, 358**  
alert statement **88, 122**  
AMUTS.PRE **366**  
And operator **53**  
Append mode **265**  
Application start-up macro **30**  
arg function **124**  
Arguments, passing to other macros **90**  
Arithmetic expressions **46–49**  
Arithmetic operators  
  Addition **47**  
  BitAnd **46, 47**  
  BitNot **46, 47**  
  BitOr **47**  
  BitXor **46, 47**  
  Division **46, 48**

Arithmetic operators, *continued*

  IntDivision **46, 48**  
  Modulo **46, 48**  
  Multiplication **46, 48**  
  Negate **46, 48**  
  Rol **46, 48**  
  Ror **46, 48**  
  Shl **46, 48**  
  Shr **46, 48**  
  Subtraction **47, 49**  
Array data type **36**  
Array declarations  
  multidimensional **68**  
  multidimensional with alternative bounds **69**  
  single dimension **68**  
  single dimension with alternative bounds **69**  
asc function **125**  
ASCII control characters **39**  
assume statement **126**

## B

backups module variable **127**  
binary function **128**  
Binary integers **38**  
BitAnd operator **46, 47**  
BitNot operator **46, 47**

BitOr operator 47  
bitstrip function 129  
BitXor operator 46, 47  
Blank lines, using 15  
Block comments 33  
Boolean data type 36  
Boolean operators 53  
Braces, using 32  
busycursor statement 130  
bye statement 131  
Byte data type 36

**C**

capture statement 132–133  
case...endcase statement 134  
CASL Macro Editor 5  
CASL overview 2  
chain statement 90, 124, 136, 176  
Char data type 36  
chdir statement 137  
Child macros 66, 90  
choice system variable 122, 138  
chr function 55, 139  
cksum function 140, 149  
class function 141  
clear statement 142  
close statement 143  
cls statement (see clear statement)  
Comments 7, 33–34  
    block 33  
    line 33–34  
    using 15  
Compatibility errors 390  
compile statement 145  
Compiler directives 56–57  
    genlabels 206  
    genlines 207  
    include 223  
    scriptdesc 298  
    trap 328  
Compiler errors 374–381  
Compiling a macro 29  
connected function (see online function)  
Connection tools 126, 167, 357, 361  
Constants 9, 37–43  
    Boolean 43  
    false 192  
    integer 37

Constants, *continued*  
    on 262  
    real 38  
    string 39  
    true 329  
Conversions, type  
    asc function 125  
    binary function 128  
    bitstrip function 129  
    chr function 139  
    class function 141  
    dehex function 161  
    detext function 166  
    enhex function 180  
    entext function 182  
    hex function 215  
    intval function 231  
    mkint function 248  
    mkstr function 249  
    octal function 260  
    str function 312  
    val function 334  
copy statement 147  
count function 148  
crc function 149  
Critical errors 386  
curday function 150  
curdir function 151  
curdrive function 152  
curhour function 153  
curminute function 154  
curmonth function 155  
cursecond function 156  
curyear function 157  
Cyclical redundancy check 149

**D**

Data capture statement 132–133  
Data type conversion 54–55  
Data types 36  
date function 158  
Date operations  
    curday function 150  
    curmonth function 155  
    curyear function 157  
    date function 158  
    overview 97  
    weekday 343



- 
- DCAT27.PRE 362
  - Decimal integers 37
  - Declarations 7
    - arrays 68
    - explicit 65
    - func...endfunc 204
    - functions 73
    - implicit 67
    - proc...endproc 277
    - procedures 70
    - public and external variables 66
    - scope rules for labels 76
    - scope rules for variables 75
  - Default keyword 134
  - definput system variable 159
  - defoutput system variable 160
  - dehex function 161
  - delete function 163
  - delete statement 162
  - description system variable 164
  - destore function 165
  - detext function 166
  - device system variable 167
  - dialogbox...enddialog statement 88, 168
  - Directives 8
  - display system variable 175
  - Display/device type 318
  - Division operator 46, 48
  - DLL errors 395
  - do statement 90, 124, 176
  - Double hyphens, line comments 34
  - drive statement 178
- E**
- Emulator or file transfer protocol errors 394
  - end statement 179
  - enhex function 180
  - enstore function 181
  - entext function 182
  - environ function 183
  - eof function 184
  - eol function 185
  - Equality operator 51
  - errclass system variable 92, 187
  - errno system variable 92, 93, 188
  - Error control
    - errclass system variable 187
    - errno system variable 188
    - error function 189
    - overview 98
    - trap compiler directive 328
  - error function 92, 189
    - trap compiler directive 187
  - Error messages 372–399
    - classes of error messages 372
    - compatibility errors 390
    - compiler errors 374–381
    - critical errors 386
    - DLL errors 395
    - emulator or file transfer protocol errors 394
    - file transfer errors 397–398
    - generic module errors 396
    - input/output errors 382–383
    - internal errors 373
    - macro execution errors 387–389
    - mathematical and range errors 384
    - missing information errors 392
    - multiple document interface errors 393
    - navigation errors 399
    - state errors 385
    - upload/download errors 391
  - Error trapping 56, 92
  - Executable files, macro 29
  - exists function 190
  - exit statement 191
  - Explicit variable declarations 65–66
  - Expressions 44–45
    - arithmetic 46–49
    - Boolean 53
    - order of evaluation 45
    - overview 9
    - relational 51
    - string 50
  - External variables 66, 91
  - EXTRA! Enterprise for Accessory Manager 2
    - connection tools 357
    - terminal tools 358
    - unsupported
      - commands 126, 131, 167, 280, 282, 302, 311, 318, 336, 340

**F**

false constant 192  
File I/O operations  
  backups module variable 127  
  capture statement 132–133  
  chdir statement 137  
  close statement 143  
  copy statement 147  
  curdir function 151  
  curdrive function 152  
  definput system variable 159  
  defoutput system variable 160  
  delete statement 162  
  drive statement 178  
  eof function 184  
  eol function 185  
  exists 190  
  filefind function 193  
  filesize function 195  
  fncheck function 196  
  fnstrip function 197  
  get statement 208  
  loc function 238  
  mkdir statement 247  
  open statement 265  
  overview 99–100  
  put statement 281  
  read line 285  
  read statement 284  
  receive statement 286  
  rename statement 287  
  rmdir statement 294  
  seek statement 300  
  send statement 301  
  write line statement 351  
  write statement 350  
File transfer errors 397–398  
File transfer protocol 280  
File transfer tools 126, 359  
filefind function 193  
filesize function 195  
fncheck function 196  
fnstrip function 197  
Focus option 172  
footer system variable 199  
for...next statement 200

Forward declarations  
  functions 74  
  procedures 71  
freemem function 202  
freetrack function 203  
func...endfunc declaration 73, 204  
Function declarations  
  argument list 73  
  forward function declaration 74  
  general description 73  
  using the forward keyword 74  
Functions  
  abs 117  
  arg 124  
  asc 125  
  binary 128  
  bitstrip 129  
  chr 139  
  cksum 140  
  class 141  
  count 148  
  crc 149  
  curday 150  
  curdir 151  
  curdrive 152  
  curhour 153  
  curminute 154  
  curmonth 155  
  cursecond 156  
  curyear 157  
  date 158  
  declaring 73  
  dehex 161  
  delete 163  
  destore 165  
  detext 166  
  enhex 180  
  enstore 181  
  entext 182  
  environ 183  
  eof 184  
  eol 185  
  error 189  
  exists 190  
  external 74  
  filefind 193  
  filesize 195  
  fncheck 196

Functions, *continued*

fnstrip 197  
 freemem 202  
 freetrack 203  
 hex 215  
 hms 219  
 inject 224  
 inkey 225  
 inscript 228  
 insert 229  
 instr 230  
 intval 231  
 left 235  
 length 236  
 loc 238  
 lowercase 239  
 max 242  
 mid 244  
 min 245  
 mkint 248  
 mkstr 249  
 name 251  
 nextchar 254  
 nextline 257  
 null 259  
 octal 260  
 online 263  
 ontime 264  
 pack 266  
 pad 267  
 quote 283  
 right 293  
 secno 299  
 session 303  
 sessname 304  
 sessno 305  
 slice 310  
 str 312  
 strip 313  
 stroke 314  
 subst 315  
 systime 316  
 time 320  
 track 326  
 upcase 332  
 val 334  
 version 335

Functions, *continued*

weekday 343  
 winchar 345  
 winsize 346  
 winsizey 347  
 winstring 348  
 winversion 349  
 xpos 352  
 ypos 353

**G**

Generic module errors 396  
 genlabels compiler directive 56, 206  
 genlines compiler directive 56, 207  
 get statement 208  
 go statement 209  
 gosub...return statement 210  
 goto statement 206, 211  
 grab statement 212  
 GreaterOrEqual operator 51  
 GreaterThan operator 51

**H**

halt statement 213  
 header system variable 214  
 hex function 54, 215  
 Hexadecimal integers 37  
 hide statement 216  
 hideallquickpads statement (see  
   unloadallquickpads statement)  
 hidequickpad statement (see unloadquickpad  
   statement)  
 hms function 219  
 homedir system variable 220  
 Host interaction  
   display system variable 175  
   match system variable 241  
   nextchar function 254  
   nextline function 257  
   nextline statement 255  
   online function 263  
   overview 84–86, 101  
   press statement 273  
   reply statement 289  
   sendbreak statement 302  
 Hyphens, double 34

**I**

ICSTOOL 357, 361  
Identifiers 35  
if...then...else statement 221  
include compiler directive 57, 72, 74, 223  
INFOConnect connection tool 357, 361  
inject function 224  
inkey function 225  
Input mode 265  
input statement 88, 227  
Input/output errors 382–383  
inscript function 228  
insert function 229  
instr function 230  
IntDivision operator 46, 48  
Integer data type 36  
Integers  
    binary 38  
    decimal 37  
    hexadecimal 37  
    kilo 38  
    octal 38  
Internal errors 373  
intval function 54, 231

**J**

jump statement (see goto statement)

**K**

Keys  
    in string constants 42  
    numeric values 225  
keys system variable 233  
Keywords 10, 58–62  
Kilo integers 38

**L**

label statement 234  
Labels  
    overview 9  
    scope rules 76  
Learn Mode 4  
left function 235  
length function 236  
LessOrEqual operator 51  
LessThan operator 51  
Limitations 2

Line comments 33–34  
    using a semicolon 34  
    using double hyphens 34  
Line continuation characters 32  
loadquickpad statement 237  
loc function 238  
lowercase function 239  
lprint statement 240

**M**

Macro elements  
    constants 9  
    expressions 9  
    keywords 10  
    labels 9  
    procedures and functions 9  
    variables 9  
Macro execution errors 387–389  
Macro management  
    chain statement 136  
    compile statement 145  
    do statement 176  
    genlabels compiler directive 206  
    genlines compiler directive 207  
    include compiler directive 223  
    inscript function 228  
    overview 102  
    quit statement 282  
    scriptdesc compiler directive 298  
    startup system variable 311  
    terminate statement 319  
    trace statement 322

Macros

    calling another macro 90  
    chaining to another macro 90  
    comments 7  
    compiling 29  
    creating 4–5  
    declarations 7  
    designing 11  
    directives 8  
    elements of 9  
    exchanging variables with other  
        macros 91  
    file types 29  
    passing arguments to other macros 90

Macros, *continued*

- running 30
- structure of 7
- types of 6

match system variable 241

Mathematical and range errors 384

## Mathematical operations

- abs function 117
- cksum function 140
- crc function 149
- intval function 231
- max function 242
- min function 245
- mkint function 248
- overview 103
- val function 334

max function 242

maximize statement 243

Messages, error 372–399

mid function 244

min function 245

minimize statement 246

Missing information errors 392

mkdir statement 247

mkint function 248

mkstr function 249

Module variables 64

- backups 127
- tabwidth 317

Modulo operator 46, 48

move statement 250

Multidimensional arrays 68

Multiple document interface errors 393

Multiple-variable declarations 65

Multiplication operator 46, 48

**N**

name function 251

Navigation errors 399

Negate operator 46, 48

netid system variable 252

new statement 253

nextchar function 254

nextline function 257

nextline statement 255

noask keyword 162

Not operator 53

null function 259

**O**

octal function 260

Octal integers 38

off constant 261

Offline macros 6

on constant 262

online function 263

Online macros 6

ontime function 264

open statement 265

Or operator 53

Output mode 265

**P**

pack function 266

pad function 267

Parent macros 66

passchar system variable 269

password system variable 270

perform statement 72, 271, 278

pop statement 272

Predefined variables 64

press statement 273

print statement 87, 275

Printer control

capture statement 132–133

footer system variable 199

grab statement 212

header system variable 214

lprint statement 240

overview 104

printer system variable 276

printer system variable 276

proc...endproc procedure declaration 70, 277

Procedure declarations 70

Procedures

argument list 70

declaring 70

external 72

forward declarations 71

overview 9

Program flow control

case...endcase statement 134

chain statement 136

do statement 176

end statement 179

exit statement 191

Program flow control, *continued*

- for...next statement 200
- freetrack function 203
- func...endfunc declaration 204
- gosub...return statement 210
- goto statement 211
- halt statement 213
- if...then...else statement 221
- label statement 234
- overview 105
- perform statement 271
- proc...endproc declaration 277
- quit statement 282
- repeat...until statement 288
- return statement 292
- terminate statement 319
- timeout system variable 321
- trace statement 322
- track function 326
- track statement 323
- wait statement 336
- watch...endwatch statement 340
- while...wend statement 344

protocol system variable 280

Protocol types 280

Public variables 66, 91

put statement 281

**Q**

quit statement 282

Quotation marks, embedded in string constants 39

quote function 283

**R**

Random mode 265

read line statement 285

read statement 284

Real data type 36

receive statement 286

Relational expressions 51

rename statement 287

repeat...until statement 85, 288

reply statement 86, 289

request statement (see receive statement)

Reserved keywords 58–62

restore statement 291

return statement 210, 292

right function 293

rmdir statement 294

Rol operator 46, 48

Ror operator 46, 48

run statement 295

**S**

Sample macros

- basic logon macro 12–15
- controlling the logon process 23–28
- verifying the host connection 16–22

save statement 296

Scope rules

- global variables 75
- labels 76
- local variables 75

script system variable 297

scriptdesc compiler directive 57, 298

secno function 299

Secret option 172

seek statement 300

Semicolon, line comments 34

send statement 301

sendbreak statement 302

Session

- creating 253
- opening 253
- start-up macro 297

session function 303

Session management

- activate statement 118
- activatesession statement 119
- assume statement 126
- bye statement 131
- description system variable 164
- device system variable 167
- go statement 209
- keys system variable 233
- name function 251
- netid system variable 252
- new statement 253
- ontime function 264
- overview 107
- password system variable 270
- protocol system variable 280
- quit statement 282
- run statement 295
- save statement 296

Session management, *continued*

- 
- script system variable 297
  - session function 303
  - sessname function 304
  - sessno function 305
  - startup system variable 311
  - terminal system variable 318
  - terminate statement 319
  - userid system variable 333
  - Session start-up macro 30
  - sessname function 304
  - sessno function 305
  - Shl operator 46, 48
  - show statement 306
  - showallquickpads statement 307
  - showquickpad statement (see loadquickpad statement)
  - Shr operator 46, 48
  - Single-dimensional arrays 68
  - Single-variable declarations 65
  - size statement 309
  - slice function 310
  - Some keyword 147
  - Source files, macro 29
  - Start-up macro, session 297
  - startup system variable 311
  - State errors 385
  - Statements 32
    - activate 118
    - activatesession 119
    - alarm 120
    - alert 122
    - assume 126
    - busycursor 130
    - bye 131
    - capture 132–133
    - case...endcase 134
    - chain 136
    - chdir 137
    - clear 142
    - close 143
    - compile 145
    - copy 147
    - delete 162
    - dialogbox...enddialog 168
    - do 176
    - drive 178
    - end 179
    - exit 191
    - for...next 200
    - get 208
    - go 209
    - gosub...return 210
    - goto 211
    - halt 213
    - hide 216
    - if...then...else 221
    - input 227
    - label 234
    - loadquickpad 237
    - lprint 240
    - maximize 243
    - minimize 246
    - mkdir 247
    - move 250
    - new 253
    - nextline 255
    - open 265
    - perform 271
    - pop 272
    - press 273
    - print 275
    - put 281
    - quit 282
    - read 284
    - read line 285
    - receive 286
    - rename 287
    - repeat...until 288
    - reply 289
    - restore 291
    - return 292
    - rmdir 294
    - run 295
    - save 296
    - seek 300
    - send 301
    - sendbreak 302
    - show 306
    - showallquickpads 307
    - size 309
    - terminate 319
    - trace 322
    - track 323
    - unloadallquickpads 330
- Statements, *continued*
- Statements, *continued*

- unloadquickpad 331
- wait 336
- watch...endwatch 340
- while...wend 344
- write 350
- write line 351
- zoom 354
- str function 54, 312
- String constants 39
  - ASCII control characters 39
  - continuing on a new line 43
  - embedded quotation marks 39
  - key names 42
- String data type 36
- String expressions 50
- String operations
  - arg function 124
  - bitstrip function 129
  - count function 148
  - dehex function 161
  - delete function 163
  - destore function 165
  - detext function 166
  - enhex function 180
  - enstore function 181
  - entext function 182
  - hex function 215
  - hms function 219
  - inject function 224
  - insert function 229
  - instr function 230
  - intval function 231
  - left function 235
  - length function 236
  - lowcase function 239
  - mid function 244
  - mkstr function 249
  - null function 259
  - overview 109
  - pack function 266
  - pad function 267
  - quote function 283
  - right function 293
  - slice function 310
  - str function 312
  - strip function 313
  - upcase 332
- String operations, *continued*
  - val function 334
  - winstring function 348
- strip function 313
- stroke function 314
- subst function 315
- Subtraction operator 49
- System variables 64
  - choice 138
  - definput 159
  - defoutput 160
  - description 164
  - device 167
  - display 175
  - errclass 187
  - errno 188
  - footer 199
  - header 214
  - keys 233
  - match 241
  - netid 252
  - passchar 269
  - password 270
  - printer 276
  - protocol 280
  - script 297
  - startup 311
  - terminal 318
  - timeout 321
  - userid 333
- systemtime function 316
- T**
  - T 27, variables 362
  - Tabstop group option 172
  - Tabstop option 172
  - tabwidth module variable 317
  - Takes keyword 277
  - Terminal emulation types 318
  - terminal system variable 318
  - Terminal tools 126, 318, 358, 362, 366
  - terminate statement 319
  - time function 320
  - Time operations
    - curhour function 153
    - curminute function 154
    - cursecond function 156
    - hms function 219
    - overview 97



- Time operations, *continued*  
  secno function 299  
  time function 320  
timeout system variable 86, 321, 336, 340  
Tools 356–369  
  connection 126, 167, 357  
  file transfer 126, 359  
  terminal 126, 318, 358  
trace statement 322  
track function 326  
track statement 203, 323  
trap compiler directive 56, 92, 189, 328  
true constant 329  
Type conversion  
  asc function 125  
  binary function 128  
  bitstrip function 129  
  chr function 139  
  class function 141  
  dehex function 161  
  detext function 166  
  enhex function 180  
  entext function 182  
  hex function 215  
  intval function 231  
  mkint function 248  
  mkstr function 249  
  octal function 260  
  overview 54–55, 111  
  str function 312  
  val function 334
- U**  
unloadallquickpads statement 330  
unloadquickpad statement 331  
upcase function 332  
Upload/download errors 391  
userid system variable 333  
UTS, variables 366
- V**  
val function 334  
Variable declarations  
  explicit 65  
  implicit 67  
  public and external 66  
Variables 9  
  backups module variable 127  
  choice system variable 138  
  definput system variable 159  
  defoutput system variable 160  
  description system variable 164  
  device system variable 167  
  display system variable 175  
  errclass system variable 187  
  errno system variable 188  
  exchanging with other macros 91  
  external 66  
  footer system variable 199  
  global 75  
  header system variable 214  
  INFOConnect connection tool 361  
  initialization 66  
  initialization values 75  
  keys system variable 233  
  local 75  
  match system variable 241  
  module 64  
  multiple-variable declarations 65  
  netid system variable 252  
  passchar system variable 269  
  password system variable 270  
  predefined 64  
  printer system variable 276  
  protocol system variable 280  
  public 66, 91  
  scope rules 75  
  script system variable 297  
  single-variable declarations 65  
  startup system variable 311  
  system 64  
  tabwidth module variable 317  
  terminal system variable 318  
  timeout system variable 321  
  userid system variable 333  
  version function 335  
VT sessions (see EXTRA! Enterprise for  
  Accessory Manager)
- W**  
wait statement 84, 336  
watch...endwatch statement 85, 340  
weekday function 343  
while...wend statement 85, 342, 344  
winchar function 345

## Index

---

### Window control

- activate statement 118
- alert statement 122
- choice system variable 138
- clear statement 142
- dialogbox...enddialog statement 168
- hide statement 216
- input statement 227
- loadquickpad statement 237
- maximize statement 243
- minimize statement 246
- move statement 250
- overview 112
- passchar system variable 269
- print statement 275
- restore statement 291
- show statement 306
- size statement 309
- tabwidth module variable 317
- unloadallquickpads statement 330
- unloadquickpad statement 331
- winchar function 345
- winsize function 346
- winsizey function 347

Window control, *continued*

- winstring function 348
- xpos function 352
- ypos function 353
- zoom statement 354

WinFTP, connection tools 357

- winsize function 346
- winsizey function 347
- winstring function 348
- winversion function 349

Word data type 36

- write line statement 351
- write statement 350

### X

- xpos function 352

XWC files 29

XWS files 29

### Y

- ypos function 353

### Z

- zoom statement 354