BCL User Manual

Basic Command Language

Version 1.5.1

October 2025

Rafa

A Tcl-inspired scripting language
with BASIC-style syntax

Designed for embedded systems and PC environments

This manual describes BCL (Basic Command Language) version 1.5.1.

BCL is inspired by Tcl 8.x but features a simplified, BASIC-style syntax designed to be beginner-friendly and suitable for both embedded systems and PC environments.

Contents

List of Tables

Listings

Chapter 1

Introduction

1.1 What is BCL?

BCL (Basic Command Language) is a lightweight, interpreted scripting language designed to combine the power of Tcl with the readability of BASIC. It was created with the following goals in mind:

- **Simplicity**: Easy to learn for beginners
- **Portability**: Runs on both PC and embedded systems
- **Flexibility**: Everything is a string, making data manipulation straightforward
- **Expressiveness**: Tcl-inspired command structure with BASIC-style keywords
- **Case-insensitivity**: Commands and keywords are not case-sensitive

Note

BCL treats **everything as a string**. Numbers, lists, and even code are all represented as text. This simplifies the language model and makes it easy to work with data from various sources.

1.2 Why BCL?

BCL fills a unique niche in the scripting language ecosystem:

- **For beginners**: The BASIC-style syntax (`IF...THEN...END`, `WHILE...DO...END`) is more intuitive than curly braces
- **For Tcl users**: Familiar command structure and evaluation model
- **For embedded systems**: Compact interpreter suitable for resource-constrained environments
- **For automation**: Simple file I/O, string processing, and system interaction

1.3 Installation and First Steps

1.3.1 Installation

BCL can be installed on various platforms. The standard distribution includes:

- The BCL interpreter executable (`bcl`)
- Standard library scripts
- Example programs
- Documentation

Refer to Appendix A for detailed installation instructions for your platform.

1.3.2 Running BCL Programs

BCL programs can be executed in two ways:

1. **Script mode**: Run a BCL script file

```
bcl myscript.bcl
```

2. **Interactive mode (REPL)**: Start the BCL interpreter without arguments

```
bcl
```

1.3.3 Your First BCL Program

Let's start with the traditional "Hello, World!" program:

Hello World

```
# This is a comment - my first BCL program
PUTS "Hello, World!"
```

Output:

```
Hello, World!
```

Explanation: The `PUTS` command prints text to the console followed by a newline. Text enclosed in double quotes is treated as a string literal.

1.3.4 Using the REPL

The REPL (Read-Eval-Print Loop) is BCL's interactive mode. It's perfect for experimenting with commands and testing small code snippets.

REPL Session

```
$ bcl
BCL> PUTS "Hello from REPL"
Hello from REPL
BCL> SET x 42
BCL> PUTS "The answer is $x"
The answer is 42
BCL> EXIT
```

Explanation: In REPL mode, you type commands and see immediate results. The `EXIT` command closes the interpreter.

Tip

Use the REPL to test BCL commands while reading this manual. It's an excellent way to learn interactively!

1.4 BCL Philosophy

Understanding BCL's core philosophy will help you write better code:

1. **Everything is a string**: All values are text. Operations determine how strings are interpreted (as numbers, lists, etc.)
2. **Commands return values**: Almost every BCL command returns a value that can be used by other commands
3. **Substitution before evaluation**: Variable values (using $) and command results (using []) are substituted before the command executes
4. **Blocks end with END**: Control structures use explicit `END` keywords instead of braces or indentation
5. **Case-insensitive**: `PUTS`, `puts`, and `Puts` are all the same command

Chapter 2

Programming Fundamentals

This chapter introduces basic programming concepts for absolute beginners using BCL.

2.1 What is a Program?

A program is a sequence of instructions that tells the computer what to do. In BCL, programs consist of commands, each on its own line or separated by semicolons.

Simple Program Structure

```
# A program with three commands
PUTS "Starting program..."
SET name "Alice"
PUTS "Hello, $name!"
```

Output:

```
Starting program...
Hello, Alice!
```

2.2 Variables: Storing Information

Variables are named containers that store values. Think of them as labeled boxes where you can put information and retrieve it later.

2.2.1 Creating Variables

Use the `SET` command to create and assign values to variables:

```
SET age 25
SET name "Bob"
SET price 19.99
```

2.2.2 Using Variables

To use a variable's value, prefix its name with a dollar sign ($):

Using Variables

```
SET username "Alice"
SET score 100

PUTS "User: $username"
PUTS "Score: $score"
```

Output:

```
User: Alice
Score: 100
```

2.3 Data Types: Everything is a String

Unlike many programming languages, BCL doesn't have separate types for numbers, text, etc. Everything is stored as a string (text). BCL automatically interprets strings as numbers when needed for calculations.

Strings as Numbers

```
SET a "10"
SET b "20"
SET sum [EXPR $a + $b]
PUTS "Sum: $sum"
```

Output:

```
Sum: 30
```

Explanation: Even though `$a` and `$b` are text, `EXPR` interprets them as numbers for arithmetic.

2.4 Comments: Documenting Your Code

Comments are notes for humans that the computer ignores. They help explain what your code does.

```
# This is a single-line comment

SET x 10   # Comments can also appear at line end
SET y 20   ;# Using semicolon before hash is also valid
```

Tip

Write comments to explain *why* your code does something, not just *what* it does. Future you will thank present you!

2.5 Basic Input and Output

2.5.1 Output: PUTS and PUTSN

The **PUTS** command displays text on the screen:

```
PUTS "This appears on the screen"
PUTS "Each PUTS starts a new line"
```

Use **PUTSN** to print without adding a newline:

PUTS vs PUTSN

```
PUTS "Line 1"
PUTS "Line 2"
PUTSN "Same "
PUTSN "line"
PUTS ""   # Empty line
```

Output:

```
Line 1
Line 2
Same line
```

2.5.2 Input: GETS

The **GETS** command reads user input from the keyboard:

Reading User Input

```
PUTS "What is your name?"
SET name [GETS stdin]
PUTS "Hello, $name!"
```

Interaction:

```
What is your name?
> Charlie
Hello, Charlie!
```

2.6 Your First Interactive Program

Let's combine what we've learned:

Interactive Greeting Program

```
# Interactive greeting program
PUTS "=== Welcome to BCL ==="
PUTS ""

# Get user's name
PUTS "Please enter your name:"
SET username [GETS stdin]
```

```

# Get user's age
PUTS "Please enter your age:"
SET age [GETS stdin]

# Display personalized greeting
PUTS ""
PUTS "Hello, $username!"
PUTS "You are $age years old."
PUTS "Welcome to the world of BCL programming!"
```

Chapter 3

Variables and Data

This chapter explores BCL's variable system in depth, including creation, modification, scope, and advanced techniques.

3.1 Creating and Modifying Variables

3.1.1 SET: The Foundation

The `SET` command is used for both creating and modifying variables.

Syntax:

```
SET variablename value          # Assign value
SET variablename                # Return current value
```

SET Examples

```
# Creating variables
SET counter 0
SET message "Hello"
SET pi 3.14159

# Modifying variables
SET counter 10
SET message "Goodbye"

# Reading values
PUTS [SET counter]   # Prints: 10
```

3.1.2 UNSET: Removing Variables

The `UNSET` command deletes a variable from memory.

UNSET Example

```
SET temp "temporary data"
PUTS $temp           # Prints: temporary data

UNSET temp
```

```
# PUTS $temp         # ERROR: variable doesn't exist
```

Warning

Accessing an unset variable causes an error. Use `INFO EXISTS` to check if a variable exists before using it.

3.2 Variable Expansion

Variable expansion means replacing `$varname` with the variable's value.

3.2.1 Basic Expansion

Variable Expansion

```
SET fruit "apple"
SET count 5

PUTS "I have $count ${fruit}s"  # Note: BCL uses only $var
PUTS "I have $count apples"
```

Output:

```
I have 5 apples
I have 5 apples
```

Note

BCL uses only the `$var` form for variable expansion. Unlike Tcl, `${var}` is not supported.

3.2.2 String Concatenation

You can concatenate strings by placing variable expansions next to each other:

Concatenation

```
SET first "John"
SET last "Doe"
SET fullname $first" "$last
PUTS $fullname  # Prints: John Doe

# Alternative with variables only
SET a "Hello"
SET b "World"
SET c $a$b
PUTS $c  # Prints: HelloWorld
```

3.3 INCR: Incrementing Numbers

`INCR` is a specialized command for incrementing (or decrementing) numeric variables.

Syntax:

```
INCR varname          # Increment by 1
INCR varname amount   # Increment by amount
```

INCR Examples

```
SET counter 10

INCR counter       # counter is now 11
INCR counter 5     # counter is now 16
INCR counter -3    # counter is now 13 (decrement)

PUTS "Counter: $counter"
```

Output:

```
Counter: 13
```

3.4 APPEND: Building Strings

`APPEND` adds text to the end of a variable, modifying it in place.

Syntax:

```
APPEND varname value1 value2 ...
```

APPEND Examples

```
SET message "Hello"
APPEND message " " "World" "!"
PUTS $message  # Prints: Hello World!

# Building strings in a loop
SET result ""
SET i 1
WHILE $i <= 5 DO
  APPEND result $i " "
  INCR i
END
PUTS $result  # Prints: 1 2 3 4 5
```

Tip

`APPEND` is more efficient than repeated `SET` operations when building large strings in loops.

3.5 Variable Scope

3.5.1 Global Variables

Variables created outside procedures are global—they can be accessed anywhere in your program.

```
SET global_var "I am global"

PROC test DO
  # Cannot access global_var here without GLOBAL declaration
END
```

3.5.2 Local Variables

Variables created with `SET` inside a procedure are local—they only exist within that procedure.

Local vs Global

```
SET outside "global"

PROC demo DO
  SET inside "local"
  PUTS "Inside proc: $inside"
END

demo
PUTS "Outside proc: $outside"
# PUTS $inside  # ERROR: inside doesn't exist here
```

Output:

```
Inside proc: local
Outside proc: global
```

3.5.3 The GLOBAL Command

To access or modify global variables from within a procedure, use **`GLOBAL`**.

Using GLOBAL

```
SET score 0

PROC add_points WITH points DO
  GLOBAL score
  INCR score $points
END

add_points 10
add_points 5
PUTS "Total score: $score"  # Prints: Total score: 15
```

3.6 Practical Examples

3.6.1 Counter Example

Hit Counter

```
SET hits 0

PROC record_hit DO
  GLOBAL hits
  INCR hits
  PUTS "Hit number $hits recorded"
END

record_hit
record_hit
record_hit
PUTS "Total hits: $hits"
```

Output:

```
Hit number 1 recorded
Hit number 2 recorded
Hit number 3 recorded
Total hits: 3
```

3.6.2 Text Builder Example

Building a Report

```
SET report ""
APPEND report "=== SYSTEM REPORT ===\n"
APPEND report "Date: " [CLOCK FORMAT [CLOCK SECONDS]] "\n"
APPEND report "Status: OK\n"
APPEND report "===================="

PUTS $report
```

Chapter 4

Expressions and Math

BCL provides powerful mathematical and logical expression evaluation through the **`EXPR`** command.

4.1 The EXPR Command

`EXPR` evaluates mathematical and logical expressions.

Syntax:

```
EXPR expression
```

Basic Arithmetic

```
SET result [EXPR 2 + 3]
PUTS $result  # Prints: 5

SET x 10
SET y 3
SET sum [EXPR $x + $y]
SET product [EXPR $x * $y]
SET quotient [EXPR $x / $y]

PUTS "Sum: $sum"           # Prints: Sum: 13
PUTS "Product: $product"   # Prints: Product: 30
PUTS "Quotient: $quotient"  # Prints: Quotient: 3.333...
```

4.2 Arithmetic Operators

Using Operators

```
# Complex expression with parentheses
SET result [EXPR (10 + 5) * 2 - 3]
PUTS $result  # Prints: 27

# Power operator
SET squared [EXPR 5 ^ 2]
PUTS "5 squared is $squared"  # Prints: 5 squared is 25
```

Table 4.1: Arithmetic Operators in BCL

Operator	Operation	Example
`+`	Addition	`EXPR 5 + 3` → 8
`-`	Subtraction	`EXPR 5 - 3` → 2
`*`	Multiplication	`EXPR 5 * 3` → 15
`/`	Division	`EXPR 5 / 2` → 2.5
`%`	Modulo (remainder)	`EXPR 5 % 3` → 2
`^`	Power	`EXPR 2 ^ 8` → 256

```

# Modulo for even/odd check
SET num 17
SET remainder [EXPR $num % 2]
IF $remainder == 0 THEN
  PUTS "$num is even"
ELSE
  PUTS "$num is odd"
END
```

4.3 Comparison Operators

Table 4.2: Comparison Operators

Operator	Meaning	Example
`=`	Equal to	`$a == 5`
`!=`	Not equal to	`$a != 5`
`<`	Less than	`$a < 10`
`<=`	Less than or equal	`$a <= 10`
`>`	Greater than	`$a > 5`
`>=`	Greater than or equal	`$a >= 5`

Comparisons

```
SET age 25

IF $age >= 18 THEN
  PUTS "Adult"
ELSE
  PUTS "Minor"
END

SET score 85
IF $score >= 90 THEN
  PUTS "Grade: A"
ELSEIF $score >= 80 THEN
```

```
  PUTS "Grade: B"
ELSEIF $score >= 70 THEN
  PUTS "Grade: C"
ELSE
  PUTS "Grade: F"
END
```

4.4 Logical Operators

Table 4.3: Logical Operators

Operator	Symbol	Example
AND	`&&`	`$a > 0 AND $a < 10`
OR	`||`	`$a == 5 OR $a == 10`
NOT	`!`	`NOT $flag`

Logical Operations

```
SET age 25
SET has_license 1

# AND operation
IF $age >= 18 AND $has_license THEN
  PUTS "Can drive"
END

# OR operation
SET day "Saturday"
IF $day == "Saturday" OR $day == "Sunday" THEN
  PUTS "It's the weekend!"
END

# NOT operation
SET raining 0
IF NOT $raining THEN
  PUTS "No umbrella needed"
END
```

4.5 Mathematical Functions

BCL includes many mathematical functions:

Trigonometry

```
SET pi 3.14159265359

# Sine of 90 degrees (pi/2 radians)
```

Table 4.4: Mathematical Functions

Function	Description
`abs(x)`	Absolute value
`sqrt(x)`	Square root
`pow(x,y)`	x raised to power y
`exp(x)`	e raised to power x
`log(x)`	Logarithm base 10
`ln(x)`	Natural logarithm (base e)
`sin(x)`	Sine (radians)
`cos(x)`	Cosine (radians)
`tan(x)`	Tangent (radians)
`asin(x)`	Arc sine
`acos(x)`	Arc cosine
`atan(x)`	Arc tangent
`hypo(x,y)`	Hypotenuse: sqrt($x^2 + y^2$)
`ceil(x)`	Round up
`floor(x)`	Round down
`round(x)`	Round to nearest integer
`int(x)`	Convert to integer
`double(x)`	Convert to floating-point
`rand()`	Random number 0.0 to 1.0
`srand(seed)`	Set random seed

```
SET angle [EXPR $pi / 2]
SET sine [EXPR sin($angle)]
PUTS [FORMAT "sin(90\textdegree) = %.4f" $sine]

# Calculate hypotenuse
SET a 3
SET b 4
SET c [EXPR hypo($a, $b)]
PUTS [FORMAT "Hypotenuse of %d and %d = %.2f" $a $b $c]
```

Output:

```
sin(90\textdegree) = 1.0000
Hypotenuse of 3 and 4 = 5.00
```

Random Numbers

```
# Generate random integer from 1 to 10
SET random [EXPR int(rand() * 10) + 1]
PUTS "Random number: $random"

# Dice roll simulator
PROC roll_dice DO
  SET roll [EXPR int(rand() * 6) + 1]
  RETURN $roll
END
```

```

SET roll1 [roll_dice]
SET roll2 [roll_dice]
PUTS "You rolled: $roll1 and $roll2"
```

4.6 Practical Examples

4.6.1 Circle Calculator

Circle Area and Circumference

```
PROC circle_stats WITH radius DO
  SET pi 3.14159265359
  SET area [EXPR $pi * $radius * $radius]
  SET circumference [EXPR 2 * $pi * $radius]

  PUTS [FORMAT "Radius: %.2f" $radius]
  PUTS [FORMAT "Area: %.2f" $area]
  PUTS [FORMAT "Circumference: %.2f" $circumference]
END

circle_stats 5.0
```

Output:

```
Radius: 5.00
Area: 78.54
Circumference: 31.42
```

4.6.2 Quadratic Equation Solver

Solving $ax^2 + bx + c$

```
PROC solve_quadratic WITH a b c DO
  # Calculate discriminant
  SET disc [EXPR $b*$b - 4*$a*$c]

  IF $disc < 0 THEN
    PUTS "No real solutions"
    RETURN
  END

  # Calculate solutions
  SET x1 [EXPR (-$b + sqrt($disc)) / (2*$a)]
  SET x2 [EXPR (-$b - sqrt($disc)) / (2*$a)]

  PUTS [FORMAT "x1 = %.4f" $x1]
  PUTS [FORMAT "x2 = %.4f" $x2]
END

# Solve x$^2$ - 5x + 6 = 0
```

```
solve_quadratic 1 -5 6
```

Output:

```
x1 = 3.0000
x2 = 2.0000
```

Chapter 5

Control Structures

Control structures let you change the order in which your program executes commands based on conditions or repetition.

5.1 IF...THEN...ELSE...END

The `IF` statement executes code conditionally.

Syntax:

```
IF condition THEN
  commands
[ELSEIF condition THEN
  commands]
[ELSE
  commands]
END
```

Simple IF

```
SET temperature 25

IF $temperature > 30 THEN
  PUTS "It's hot!"
ELSEIF $temperature > 20 THEN
  PUTS "It's warm"
ELSEIF $temperature > 10 THEN
  PUTS "It's cool"
ELSE
  PUTS "It's cold!"
END
```

Output:

```
It's warm
```

Nested IF

```
SET age 25
SET student 1

```

```
IF $age < 18 THEN
  PUTS "Minor - discounted ticket"
ELSE
  IF $student THEN
    PUTS "Adult student - discounted ticket"
  ELSE
    PUTS "Regular adult ticket"
  END
END
```

5.2 WHILE Loops

`WHILE` repeats code as long as a condition is true.

Syntax:

```
WHILE condition DO
  commands
END
```

Countdown

```
SET count 5

WHILE $count > 0 DO
  PUTS "Countdown: $count"
  INCR count -1
END

PUTS "Liftoff!"
```

Output:

```
Countdown: 5
Countdown: 4
Countdown: 3
Countdown: 2
Countdown: 1
Liftoff!
```

Input Validation

```
SET valid 0

WHILE NOT $valid DO
  PUTS "Enter a number between 1 and 10:"
  SET input [GETS stdin]

  IF $input >= 1 AND $input <= 10 THEN
    SET valid 1
    PUTS "Thank you! You entered $input"
  ELSE
```

```
    PUTS "Invalid input. Try again."
  END
END
```

5.3 FOR Loops

BCL supports two styles of **FOR** loops.

5.3.1 FOR with Explicit Variable

FOR Loop - Explicit Variable

```
FOR [SET i 1] TO $i <= 10 DO
  PUTS "Iteration $i"
  INCR i
END
```

5.3.2 FOR with Internal Counter

FOR Loop - Internal Counter

```
FOR 1 TO 10 DO
  PUTS "Number: $__FOR"
END

# With STEP
FOR 0 TO 100 STEP 10 DO
  PUTS "Value: $__FOR"
END
```

Output:

```
Number: 1
Number: 2
...
Number: 10
Value: 0
Value: 10
...
Value: 100
```

Note

The internal counter variable is named `$__FOR` and is automatically created by BCL.

5.4 FOREACH Loops

FOREACH iterates over list elements.

Syntax:

```
FOREACH variable IN list DO
  commands
END
```

Iterating Lists

```
SET colors [LIST red green blue yellow]

FOREACH color IN $colors DO
  PUTS "Color: $color"
END
```

Output:

```
Color: red
Color: green
Color: blue
Color: yellow
```

Processing Data

```
SET scores [LIST 85 92 78 95 88]
SET total 0
SET count 0

FOREACH score IN $scores DO
  SET total [EXPR $total + $score]
  INCR count
END

SET average [EXPR $total / $count]
PUTS [FORMAT "Average score: %.2f" $average]
```

Output:

```
Average score: 87.60
```

5.5 SWITCH Statement

`SWITCH` selects one of many code blocks to execute.

Syntax:

```
SWITCH expression DO
  CASE value1
    commands
  CASE value2
    commands
  DEFAULT
    commands
END
```

SWITCH Example

```
SET day "Tuesday"

SWITCH $day DO
  CASE "Monday"
    PUTS "Start of work week"
  CASE "Tuesday"
    PUTS "Second day"
  CASE "Wednesday"
    PUTS "Midweek"
  CASE "Thursday"
    PUTS "Almost Friday"
  CASE "Friday"
    PUTS "Last work day!"
  DEFAULT
    PUTS "Weekend!"
END
```

Output:

```
Second day
```

Menu System

```
PUTS "Select an option:"
PUTS "1. New file"
PUTS "2. Open file"
PUTS "3. Save file"
PUTS "4. Exit"

SET choice [GETS stdin]

SWITCH $choice DO
  CASE "1"
    PUTS "Creating new file..."
  CASE "2"
    PUTS "Opening file..."
  CASE "3"
    PUTS "Saving file..."
  CASE "4"
    PUTS "Goodbye!"
    EXIT
  DEFAULT
    PUTS "Invalid choice"
END
```

5.6 BREAK and CONTINUE

5.6.1 BREAK: Exit Loop Early

`BREAK` terminates the innermost loop immediately.

Using BREAK

```
# Search for a number
SET target 7
SET found 0

FOR 1 TO 10 DO
  IF $__FOR == $target THEN
    SET found $__FOR
    BREAK
  END
END

IF $found THEN
  PUTS "Found $target"
ELSE
  PUTS "Not found"
END
```

5.6.2 CONTINUE: Skip to Next Iteration

`CONTINUE` skips the rest of the current iteration and goes to the next one.

Using CONTINUE

```
# Print odd numbers only
FOR 1 TO 10 DO
  SET num $__FOR
  SET remainder [EXPR $num % 2]

  IF $remainder == 0 THEN
    CONTINUE  # Skip even numbers
  END

  PUTS $num
END
```

Output:

```
1
3
5
7
9
```

5.7 EXIT: Terminating the Program

`EXIT` terminates the entire program (or REPL session).

```
EXIT           # Exit with code 0
EXIT 1         # Exit with code 1 (error)
```

5.8 Complete Examples

5.8.1 Number Guessing Game

Guessing Game

```
# Generate random number 1-100
SET secret [EXPR int(rand() * 100) + 1]
SET guesses 0
SET found 0

PUTS "I'm thinking of a number between 1 and 100"

WHILE NOT $found DO
  PUTS "Enter your guess:"
  SET guess [GETS stdin]
  INCR guesses

  IF $guess == $secret THEN
    SET found 1
    PUTS "Correct! You won in $guesses guesses!"
  ELSEIF $guess < $secret THEN
    PUTS "Too low!"
  ELSE
    PUTS "Too high!"
  END
END
```

5.8.2 FizzBuzz

FizzBuzz Classic

```
FOR 1 TO 100 DO
  SET num $__FOR
  SET by3 [EXPR $num % 3]
  SET by5 [EXPR $num % 5]

  IF $by3 == 0 AND $by5 == 0 THEN
    PUTS "FizzBuzz"
  ELSEIF $by3 == 0 THEN
    PUTS "Fizz"
  ELSEIF $by5 == 0 THEN
    PUTS "Buzz"
  ELSE
    PUTS $num
  END
END
```

Chapter 6

Procedures (Functions)

Procedures (also called functions in other languages) let you package code into reusable blocks.

6.1 Defining Procedures

Syntax with parameters:

```
PROC name WITH param1 param2 ... DO
  commands
  RETURN value
END
```

Syntax without parameters:

```
PROC name DO
  commands
  RETURN value
END
```

Note

When a procedure has no parameters, the `WITH` keyword can be omitted.

Procedure with Parameter

```
PROC greet WITH name DO
  PUTS "Hello, $name!"
END

greet "Alice"
greet "Bob"
```

Output:

```
Hello, Alice!
Hello, Bob!
```

Procedure without Parameters

```
PROC show_version DO
  PUTS "BCL Interpreter v1.5.0"
END

show_version
```

Output:

```
BCL Interpreter v1.5.0
```

Note

Procedures are invoked by their name alone—no `CALL` keyword is needed.

6.2 Parameters

6.2.1 Fixed Parameters

Parameters are declared in the `WITH` clause and are accessible as variables inside the procedure.

Multiple Parameters

```
PROC add WITH a b DO
  SET result [EXPR $a + $b]
  RETURN $result
END

SET sum [add 5 3]
PUTS "5 + 3 = $sum"
```

Output:

```
5 + 3 = 8
```

6.2.2 Optional Parameters

Optional parameters are prefixed with `@`. They may or may not be provided when calling the procedure.

Optional Parameters

```
PROC greet WITH name @title DO
  IF [INFO EXISTS title] THEN
    PUTS "Hello, $title $name"
  ELSE
    PUTS "Hello, $name"
  END
END

greet "Smith"
greet "Smith" "Dr."
```

Output:

```
Hello, Smith
Hello, Dr. Smith
```

Tip

Use **INFO EXISTS** to check if an optional parameter was provided.

6.3 Return Values

The **RETURN** command exits a procedure and optionally returns a value.

Returning Values

```
PROC square WITH n DO
  SET result [EXPR $n * $n]
  RETURN $result
END

PROC is_even WITH n DO
  SET remainder [EXPR $n % 2]
  IF $remainder == 0 THEN
    RETURN 1   # true
  ELSE
    RETURN 0   # false
  END
END

SET s [square 7]
PUTS "7 squared = $s"

IF [is_even 10] THEN
  PUTS "10 is even"
END
```

Output:

```
7 squared = 49
10 is even
```

Note

If **RETURN** is called without a value, or if a procedure reaches its **END** without returning, it returns an empty string.

6.4 Variable Scope in Procedures

6.4.1 Local Variables

Variables created with **SET** inside a procedure are local—they exist only within that procedure.

Local Variables

```
PROC calculate WITH x DO
  SET double [EXPR $x * 2]   # local variable
  SET triple [EXPR $x * 3]   # local variable
  PUTS "Inside: double=$double, triple=$triple"
END

calculate 5
# PUTS $double  # ERROR: double doesn't exist here
```

6.4.2 Accessing Global Variables

Use `GLOBAL` to access or modify global variables from within a procedure.

Global Variables in Procedures

```
SET counter 0

PROC increment WITH amount DO
  GLOBAL counter
  INCR counter $amount
  PUTS "Counter is now: $counter"
END

increment 5
increment 3
PUTS "Final counter: $counter"
```

Output:

```
Counter is now: 5
Counter is now: 8
Final counter: 8
```

6.5 Recursive Procedures

Procedures can call themselves—this is called recursion.

Factorial (Recursive)

```
PROC factorial WITH n DO
  IF $n <= 1 THEN
    RETURN 1
  ELSE
    SET prev [factorial [EXPR $n - 1]]
    RETURN [EXPR $n * $prev]
  END
END

PUTS "5! = [factorial 5]"
PUTS "10! = [factorial 10]"
```

Output:

```
5! = 120
10! = 3628800
```

Warning

Recursive procedures must have a base case (termination condition) to avoid infinite recursion!

Fibonacci (Recursive)

```
PROC fib WITH n DO
  IF $n <= 1 THEN
    RETURN $n
  ELSE
    SET n1 [EXPR $n - 1]
    SET n2 [EXPR $n - 2]
    SET f1 [fib $n1]
    SET f2 [fib $n2]
    RETURN [EXPR $f1 + $f2]
  END
END

# Print first 10 Fibonacci numbers
FOR 0 TO 9 DO
  SET f [fib $__FOR]
  PUTS "fib($__FOR) = $f"
END
```

6.6 Practical Examples

6.6.1 Temperature Converter

Celsius to Fahrenheit

```
PROC celsius_to_fahrenheit WITH celsius DO
  SET fahrenheit [EXPR ($celsius * 9.0 / 5.0) + 32]
  RETURN $fahrenheit
END

PROC fahrenheit_to_celsius WITH fahrenheit DO
  SET celsius [EXPR ($fahrenheit - 32) * 5.0 / 9.0]
  RETURN $celsius
END

SET c 25
SET f [celsius_to_fahrenheit $c]
PUTS [FORMAT "%d\textdegreeC = %.1f\textdegreeF" $c $f]

SET f2 100
SET c2 [fahrenheit_to_celsius $f2]
PUTS [FORMAT "%d\textdegreeF = %.1f\textdegreeC" $f2 $c2]
```

Output:

```
25\textdegreeC = 77.0\textdegreeF
100\textdegreeF = 37.8\textdegreeC
```

6.6.2 String Utilities

String Helper Procedures

```
PROC reverse_string WITH str DO
  RETURN [STRING REVERSE $str]
END

PROC capitalize WITH str DO
  SET first [STRING INDEX $str 0]
  SET rest [STRING RANGE $str 1 end]
  SET first_upper [STRING TOUPPER $first]
  SET rest_lower [STRING TOLOWER $rest]
  RETURN [STRING CAT $first_upper $rest_lower]
END

SET text "hello world"
PUTS "Original: $text"
PUTS "Reversed: [reverse_string $text]"
PUTS "Capitalized: [capitalize $text]"
```

Output:

```
Original: hello world
Reversed: dlrow olleh
Capitalized: Hello world
```

Chapter 7

Lists

Lists are collections of values stored in a single variable. BCL provides comprehensive list manipulation commands.

7.1 Creating Lists

7.1.1 LIST Command

Creating Lists

```
SET numbers [LIST 1 2 3 4 5]
SET colors [LIST red green blue]
SET mixed [LIST "hello world" 42 3.14]

PUTS $numbers
PUTS $colors
```

Output:

```
1 2 3 4 5
red green blue
```

7.1.2 SPLIT Command

`SPLIT` creates a list by splitting a string on a separator.

Splitting Strings

```
SET csv "apple,banana,cherry,date"
SET fruits [SPLIT $csv ","]

FOREACH fruit IN $fruits DO
  PUTS "Fruit: $fruit"
END
```

Output:

```
Fruit: apple
Fruit: banana
Fruit: cherry
```

```
Fruit: date
```

7.2 Accessing List Elements

7.2.1 LINDEX: Get Element by Index

Accessing Elements

```
SET names [LIST Alice Bob Charlie Diana]

SET first [LINDEX $names 0]
SET second [LINDEX $names 1]
SET last [LINDEX $names 3]

PUTS "First: $first"
PUTS "Second: $second"
PUTS "Last: $last"
```

Output:

```
First: Alice
Second: Bob
Last: Diana
```

7.2.2 LRANGE: Get Sublist

Extracting Sublists

```
SET numbers [LIST 10 20 30 40 50 60]

SET first_three [LRANGE $numbers 0 2]
SET last_two [LRANGE $numbers 4 5]
SET middle [LRANGE $numbers 2 4]

PUTS "First three: $first_three"
PUTS "Last two: $last_two"
PUTS "Middle: $middle"
```

Output:

```
First three: 10 20 30
Last two: 50 60
Middle: 30 40 50
```

7.2.3 LLENGTH: Get List Length

List Length

```
SET items [LIST pen paper pencil eraser]
SET count [LLENGTH $items]

PUTS "The list has $count items"

# Iterate using length
SET i 0
WHILE $i < $count DO
  SET item [LINDEX $items $i]
  PUTS "Item $i: $item"
  INCR i
END
```

7.3 Modifying Lists

7.3.1 LAPPEND: Add to End

Appending to Lists

```
SET fruits [LIST apple banana]
SET fruits [LAPPEND $fruits cherry]
SET fruits [LAPPEND $fruits date elderberry]

PUTS $fruits
```

Output:

```
apple banana cherry date elderberry
```

7.3.2 LINSERT: Insert at Position

Inserting Elements

```
SET numbers [LIST 1 2 4 5]
# Insert 3 at index 2
SET numbers [LINSERT $numbers 2 3]

PUTS $numbers  # Prints: 1 2 3 4 5
```

7.3.3 LREPLACE: Replace Range

Replacing Elements

```
SET letters [LIST a b c d e]

# Replace index 1-2 with X Y
SET letters [LREPLACE $letters 1 2 X Y]
PUTS $letters  # Prints: a X Y d e

# Delete elements (no replacement)
SET letters [LREPLACE $letters 1 2]
PUTS $letters  # Prints: a d e
```

7.4 Sorting and Searching

7.4.1 LSORT: Sort Lists

Sorting

```
SET unsorted [LIST zebra apple monkey dog cat]
SET sorted [LSORT $unsorted]

PUTS "Unsorted: $unsorted"
PUTS "Sorted: $sorted"
```

Output:

```
Unsorted: zebra apple monkey dog cat
Sorted: apple cat dog monkey zebra
```

Note

`LSORT` performs ASCII sorting, which is case-sensitive. Uppercase letters come before lowercase.

7.4.2 LSEARCH: Find Element

Searching Lists

```
SET fruits [LIST apple banana cherry date]

SET idx1 [LSEARCH $fruits "banana"]
SET idx2 [LSEARCH $fruits "grape"]

PUTS "Index of banana: $idx1"   # 1
PUTS "Index of grape: $idx2"    # -1 (not found)

IF $idx1 != -1 THEN
  PUTS "Found banana at position $idx1"
END
```

7.5 List Operations

7.5.1 JOIN: List to String

Joining Lists

```
SET words [LIST Hello World from BCL]

SET sentence [JOIN $words " "]
SET csv [JOIN $words ","]

PUTS $sentence   # Hello World from BCL
PUTS $csv        # Hello,World,from,BCL
```

7.5.2 CONCAT: Combine Lists

Concatenating Lists

```
SET list1 [LIST 1 2 3]
SET list2 [LIST 4 5 6]
SET list3 [LIST 7 8 9]

SET combined [CONCAT $list1 $list2 $list3]
PUTS $combined  # 1 2 3 4 5 6 7 8 9
```

7.6 Practical Examples

7.6.1 Shopping List Manager

Shopping List

```
SET shopping_list [LIST]

PROC add_item WITH item DO
  GLOBAL shopping_list
  SET shopping_list [LAPPEND $shopping_list $item]
  PUTS "Added: $item"
END

PROC show_list DO
  GLOBAL shopping_list
  SET count [LLENGTH $shopping_list]

  IF $count == 0 THEN
    PUTS "Shopping list is empty"
    RETURN
  END

  PUTS "Shopping List ($count items):"
  SET i 0
  FOREACH item IN $shopping_list DO
    INCR i
    PUTS "  $i. $item"
  END
END

add_item "Milk"
add_item "Bread"
add_item "Eggs"
show_list
```

Output:

```
Added: Milk
Added: Bread
Added: Eggs
Shopping List (3 items):
  1. Milk
  2. Bread
  3. Eggs
```

7.6.2 Grade Calculator

Calculate Average Grade

```
SET grades [LIST 85 92 78 90 88 95]

# Calculate sum
SET sum 0
FOREACH grade IN $grades DO
  SET sum [EXPR $sum + $grade]
END

# Calculate average
SET count [LLENGTH $grades]
SET average [EXPR $sum / $count]

# Find min and max
SET sorted [LSORT $grades]
SET min [LINDEX $sorted 0]
SET max [LINDEX $sorted [EXPR $count - 1]]

PUTS [FORMAT "Count: %d" $count]
PUTS [FORMAT "Average: %.2f" $average]
PUTS [FORMAT "Minimum: %d" $min]
PUTS [FORMAT "Maximum: %d" $max]
```

Output:

```
Count: 6
Average: 88.00
Minimum: 78
Maximum: 95
```

Chapter 8

String Manipulation

Strings (text) are the foundation of BCL. Since everything in BCL is a string, understanding how to manipulate text is essential. This chapter covers all the tools BCL provides for working with strings.

8.1 What Are Strings?

A string is simply a sequence of characters—letters, numbers, symbols, spaces, etc. In BCL, strings can be:

- Single words: `hello`
- Sentences: `Hello, World!`
- Numbers stored as text: `42` or `3.14`
- Empty: `""` (a string with no characters)

Creating Strings

```
# Simple strings
SET greeting "Hello"
SET message "Welcome to BCL programming!"

# Strings with numbers
SET year "2025"
SET price "19.99"

# Empty string
SET empty ""

# Strings with special characters
SET symbols "!@#$%^&*()"
```

Note

Remember: In BCL, everything is a string. Even the number `42` is stored as the two-character string "4" and "2".

8.2 The STRING Command

The **STRING** command is your Swiss Army knife for text manipulation. It has many subcommands, each performing a specific operation.

General Syntax:

```
STRING subcommand arguments...
```

8.2.1 STRING LENGTH - Measuring Text

`STRING LENGTH` tells you how many characters are in a string.

Syntax:

```
STRING LENGTH string
```

String Length

```
SET word "hello"
SET len [STRING LENGTH $word]
PUTS "The word '$word' has $len characters"

SET sentence "This is a test"
PUTS "Length: [STRING LENGTH $sentence]"

# Empty string has length 0
SET empty ""
PUTS "Empty string length: [STRING LENGTH $empty]"
```

Output:

```
The word 'hello' has 5 characters
Length: 14
Empty string length: 0
```

Tip

The length includes spaces and punctuation. "hi!" has length 3, not 2.

8.2.2 STRING INDEX - Getting Single Characters

`STRING INDEX` extracts one character from a specific position.

Syntax:

```
STRING INDEX string position
```

Note

Positions start at 0! The first character is at position 0, the second at position 1, etc.

Extracting Characters

```
SET text "HELLO"

# Get first character (position 0)
SET first [STRING INDEX $text 0]
PUTS "First: $first"   # H

# Get third character (position 2)
SET third [STRING INDEX $text 2]
PUTS "Third: $third"   # L

# Get last character
SET len [STRING LENGTH $text]
SET last_pos [EXPR $len - 1]
SET last [STRING INDEX $text $last_pos]
PUTS "Last: $last"   # O

# You can use 'end' for the last character
SET last2 [STRING INDEX $text end]
PUTS "Also last: $last2"   # O
```

Output:

```
First: H
Third: L
Last: O
Also last: O
```

8.2.3 STRING RANGE - Extracting Substrings

`STRING RANGE` extracts a portion of a string from one position to another.

Syntax:

```
STRING RANGE string start end
```

Substring Extraction

```
SET text "Hello, World!"

# Get first 5 characters (positions 0-4)
SET hello [STRING RANGE $text 0 4]
PUTS $hello   # Hello

# Get "World" (positions 7-11)
SET world [STRING RANGE $text 7 11]
PUTS $world   # World

# Get from position 7 to the end
SET rest [STRING RANGE $text 7 end]
PUTS $rest   # World!

# Get last 6 characters
SET last [STRING RANGE $text end-5 end]
```

```
PUTS $last   # World!
```

Output:

```
Hello
World
World!
World!
```

Tip

Use `end` to refer to the last position, and `end-N` to count backwards from the end.

8.2.4 STRING TOUPPER and TOLOWER - Changing Case

These commands convert strings to uppercase or lowercase.

Syntax:

```
STRING TOUPPER string
STRING TOLOWER string
```

Case Conversion

```
SET text "Hello World"

SET upper [STRING TOUPPER $text]
PUTS $upper   # HELLO WORLD

SET lower [STRING TOLOWER $text]
PUTS $lower   # hello world

# Useful for case-insensitive comparisons
SET input "YES"
SET normalized [STRING TOLOWER $input]

IF $normalized == "yes" THEN
  PUTS "User said yes!"
END
```

Output:

```
HELLO WORLD
hello world
User said yes!
```

8.2.5 STRING TRIM - Removing Whitespace

`STRING TRIM` removes spaces, tabs, and newlines from the beginning and/or end of a string.

Syntax:

```
STRING TRIM string                 # Remove from both ends
STRING TRIMLEFT string             # Remove from left only
STRING TRIMRIGHT string            # Remove from right only
STRING TRIM string characters   # Remove specific characters
```

Trimming Whitespace

```
# User input often has extra spaces
SET input "   hello   "

SET clean [STRING TRIM $input]
PUTS "[$clean]"  # [hello]

# Trim only from left
SET left [STRING TRIMLEFT $input]
PUTS "[$left]"  # [hello   ]

# Trim only from right
SET right [STRING TRIMRIGHT $input]
PUTS "[$right]"  # [   hello]

# Trim specific characters
SET text "***Hello***"
SET trimmed [STRING TRIM $text "*"]
PUTS $trimmed  # Hello
```

Output:

```
[hello]
[hello   ]
[   hello]
Hello
```

8.2.6 STRING COMPARE - Comparing Strings

`STRING COMPARE` compares two strings and returns:

- -1 if first string comes before second (alphabetically)
- 0 if strings are identical
- 1 if first string comes after second

Syntax:

```
STRING COMPARE string1 string2
STRING COMPARE -nocase string1 string2  # Ignore case
```

String Comparison

```
# Exact comparison
SET result [STRING COMPARE "apple" "banana"]
PUTS $result  # -1 (apple comes before banana)

SET result [STRING COMPARE "zoo" "ant"]
PUTS $result  # 1 (zoo comes after ant)

```

```
SET result [STRING COMPARE "hello" "hello"]
PUTS $result  # 0 (identical)

# Case-insensitive comparison
SET r1 [STRING COMPARE "Hello" "hello"]
PUTS "Case-sensitive: $r1"  # 1 (different)

SET r2 [STRING COMPARE -nocase "Hello" "hello"]
PUTS "Case-insensitive: $r2"  # 0 (same)
```

Output:

```
-1
1
0
Case-sensitive: 1
Case-insensitive: 0
```

8.2.7 STRING FIRST and LAST - Finding Substrings

These commands find the position of a substring within a string.

Syntax:

```
STRING FIRST substring string [startpos]
STRING LAST substring string [startpos]
```

Finding Substrings

```
SET text "hello world, hello BCL"

# Find first occurrence of "hello"
SET pos [STRING FIRST "hello" $text]
PUTS "First 'hello' at position: $pos"  # 0

# Find last occurrence of "hello"
SET pos [STRING LAST "hello" $text]
PUTS "Last 'hello' at position: $pos"  # 13

# Search starting from position 5
SET pos [STRING FIRST "hello" $text 5]
PUTS "Next 'hello' after pos 5: $pos"  # 13

# Not found returns -1
SET pos [STRING FIRST "goodbye" $text]
IF $pos == -1 THEN
  PUTS "'goodbye' not found"
END
```

Output:

```
First 'hello' at position: 0
Last 'hello' at position: 13
Next 'hello' after pos 5: 13
'goodbye' not found
```

8.2.8 STRING REPLACE - Replacing Text

`STRING REPLACE` replaces part of a string with new text.

Syntax:

```
STRING REPLACE string start end newtext
```

Replacing Parts of Strings

```
SET text "Hello World"

# Replace "World" (positions 6-10) with "BCL"
SET new [STRING REPLACE $text 6 10 "BCL"]
PUTS $new  # Hello BCL

# Replace first word
SET new [STRING REPLACE $text 0 4 "Goodbye"]
PUTS $new  # Goodbye World

# Delete part of string (replace with empty)
SET text "Hello, World!"
SET new [STRING REPLACE $text 5 6 ""]
PUTS $new  # Hello World!
```

Output:

```
Hello BCL
Goodbye World
Hello World!
```

8.2.9 STRING REVERSE - Reversing Text

`STRING REVERSE` reverses the order of characters in a string.

Syntax:

```
STRING REVERSE string
```

Reversing Strings

```
SET text "hello"
SET reversed [STRING REVERSE $text]
PUTS $reversed  # olleh

SET text "racecar"
SET rev [STRING REVERSE $text]
IF $text == $rev THEN
  PUTS "'$text' is a palindrome!"
END
```

Output:

```
olleh
'racecar' is a palindrome!
```

8.2.10 STRING MATCH - Pattern Matching

`STRING MATCH` checks if a string matches a pattern with wildcards.

Patterns:

- `*` - matches any sequence of characters
- `?` - matches any single character
- `[abc]` - matches any character in brackets

Syntax:

```
STRING MATCH pattern string
STRING MATCH -nocase pattern string  # Ignore case
```

Pattern Matching

```
# Match with wildcards
IF [STRING MATCH "*.txt" "document.txt"] THEN
  PUTS "It's a text file"
END

# Match any 3-letter word
IF [STRING MATCH "???" "cat"] THEN
  PUTS "Three letter word"
END

# Match email pattern
SET email "user@example.com"
IF [STRING MATCH "*@*.*" $email] THEN
  PUTS "Looks like an email"
END

# Character sets
IF [STRING MATCH "\[0-9\]*" "123abc"] THEN
  PUTS "Starts with a digit"
END
```

Output:

```
It's a text file
Three letter word
Looks like an email
Starts with a digit
```

8.3 FORMAT and SCAN - Formatted Text

8.3.1 FORMAT: Creating Formatted Output

`FORMAT` creates formatted strings, similar to printf in C.

Common Format Specifiers:

Width and Precision:

- `%10s` - String with minimum width 10 (right-aligned)

Table 8.1: FORMAT Specifiers

Specifier	Description
`%s`	String
`%d`	Integer (decimal)
`%f`	Floating-point number
`%x`	Hexadecimal
`%o`	Octal
`%c`	Character (from ASCII code)
`%%`	Literal % sign

- `%-10s` - String with minimum width 10 (left-aligned)
- `%.2f` - Floating-point with 2 decimal places
- `%8.2f` - Width 8, 2 decimal places

FORMAT Examples

```
SET name "Alice"
SET age 30
SET height 1.68
SET score 95.5

# Basic formatting
PUTS [FORMAT "Name: %s, Age: %d" $name $age]

# Floating-point precision
PUTS [FORMAT "Height: %.2f meters" $height]
PUTS [FORMAT "Score: %.1f%%" $score]

# Width and alignment
PUTS [FORMAT "%10s | %5d | %6.2f" $name $age $height]
PUTS [FORMAT "%-10s | %-5d | %-6.2f" $name $age $height]

# Creating tables
PUTS [FORMAT "%-10s %8s %8s" "Name" "Age" "Height"]
PUTS [FORMAT "%-10s %8d %8.2f" "Alice" 30 1.68]
PUTS [FORMAT "%-10s %8d %8.2f" "Bob" 25 1.75]

# Numbers in different bases
SET num 255
PUTS [FORMAT "Decimal: %d" $num]
PUTS [FORMAT "Hex: %x" $num]
PUTS [FORMAT "Octal: %o" $num]
```

Output:

```
Name: Alice, Age: 30
Height: 1.68 meters
Score: 95.5%
     Alice |    30 |   1.68
Alice      | 30    | 1.68
```

```
Name              Age   Height
Alice              30     1.68
Bob                25     1.75
Decimal: 255
Hex: ff
Octal: 377
```

8.3.2 SCAN: Parsing Formatted Input

SCAN is the opposite of FORMAT—it extracts values from a formatted string.

Syntax:

```
SCAN string format var1 var2 ...
```

SCAN Examples

```
# Parse structured data
SET data "John 25 180.5"
SCAN $data "%s %d %f" name age height

PUTS "Name: $name"
PUTS "Age: $age"
PUTS "Height: $height"

# Parse date
SET date "2025-10-22"
SCAN $date "%d-%d-%d" year month day
PUTS "Year: $year, Month: $month, Day: $day"

# Parse key=value pairs
SET config "timeout=30"
SCAN $config "%\[^=\]=%d" key value
PUTS "Key: $key, Value: $value"

# Count items parsed
SET count [SCAN "42 3.14 hello" "%d %f %s" a b c]
PUTS "Parsed $count items"
```

Output:

```
Name: John
Age: 25
Height: 180.5
Year: 2025, Month: 10, Day: 22
Key: timeout, Value: 30
Parsed 3 items
```

8.4 Practical Examples

8.4.1 Email Validator

Simple Email Validation

```
PROC is_valid_email WITH email DO
  # Check for @ symbol
  SET at_pos [STRING FIRST "@" $email]
  IF $at_pos == -1 THEN
    RETURN 0
  END

  # Check for dot after @
  SET dot_pos [STRING FIRST "." $email $at_pos]
  IF $dot_pos == -1 THEN
    RETURN 0
  END

  # Basic pattern match
  IF [STRING MATCH "*@*.*" $email] THEN
    RETURN 1
  END

  RETURN 0
END

# Test the validator
SET emails [LIST "user@example.com" "invalid.email"
   "test@domain.co.uk"]
FOREACH email IN $emails DO
  IF [is_valid_email $email] THEN
    PUTS "$email - VALID"
  ELSE
    PUTS "$email - INVALID"
  END
END
```

Output:

```
user@example.com - VALID
invalid.email - INVALID
test@domain.co.uk - VALID
```

8.4.2 Text Formatter

Centering Text

```
PROC center_text WITH text width DO
  SET len [STRING LENGTH $text]

  # Text is already too long
  IF $len >= $width THEN
    RETURN $text
  END

  # Calculate padding
  SET total_pad [EXPR $width - $len]
  SET left_pad [EXPR $total_pad / 2]

  # Create padding string
  SET padding ""
  FOR 1 TO $left_pad DO
    APPEND padding " "
  END

  # Return centered text
  RETURN $padding$text
END

# Create a title
SET title "BCL Manual"
SET line [center_text $title 40]
PUTS $line

SET border [STRING REPEAT "=" 40]
PUTS $border
```

Output:

```
               BCL Manual
========================================
```

8.4.3 Word Counter

Counting Words

```
PROC count_words WITH text DO
  # Trim whitespace
  SET clean [STRING TRIM $text]

  # Empty string has 0 words
  IF [STRING LENGTH $clean] = 0 THEN
    RETURN 0
  END

  # Count spaces and add 1
  SET count 1
  SET pos 0
  WHILE 1 DO
    SET pos [STRING FIRST " " $clean $pos]
    IF $pos == -1 THEN
      BREAK
    END
    INCR count
    INCR pos
  END

  RETURN $count
END

SET sentence "The quick brown fox jumps"
SET wc [count_words $sentence]
PUTS "Words: $wc"

SET text "  Multiple   spaces   between  "
PUTS "Words in '$text': [count_words $text]"
```

Output:

```
Words: 5
Words in '  Multiple   spaces   between  ': 4
```

8.4.4 Password Strength Checker

Check Password Strength

```
PROC check_password WITH pass DO
  SET len [STRING LENGTH $pass]

  # Too short
  IF $len < 8 THEN
    PUTS "Weak: Too short (minimum 8 characters)"
    RETURN
  END

  # Check for digits
  SET has_digit 0
  FOR 0 TO $len-1 DO
    SET char [STRING INDEX $pass $__FOR]
    IF [STRING MATCH "\[0-9\]" $char] THEN
      SET has_digit 1
      BREAK
    END
  END

  # Check for uppercase
  SET upper [STRING TOUPPER $pass]
  SET has_upper [EXPR $pass != $upper]

  # Check for lowercase
  SET lower [STRING TOLOWER $pass]
  SET has_lower [EXPR $pass != $lower]

  # Calculate strength
  SET strength 0
  IF $len >= 8 THEN
    INCR strength
  END
  IF $len >= 12 THEN
    INCR strength
  END
  IF $has_digit THEN
    INCR strength
  END
  IF $has_upper THEN
    INCR strength
  END
  IF $has_lower THEN
    INCR strength
  END

  # Report
  IF $strength <= 2 THEN
    PUTS "Weak password"
  ELSEIF $strength <= 3 THEN
    PUTS "Medium password"
  ELSE
    PUTS "Strong password"
  END
END

check_password "hello"
check_password "hello123"
check_password "Hello123"
check_password "MyP@ssw0rd2025"
```

Output:

```
Weak: Too short (minimum 8 characters)
Medium password
Strong password
Strong password
```

8.5 Common String Patterns

8.5.1 Building Strings Efficiently

String Building Techniques

```
# Method 1: Using APPEND (efficient for loops)
SET result ""
FOR 1 TO 5 DO
  APPEND result "Line " $__FOR "\n"
END
PUTS $result

# Method 2: Using STRING CAT
SET str1 "Hello"
SET str2 "World"
SET combined [STRING CAT $str1 " " $str2]
PUTS $combined

```

```
# Method 3: Building with FORMAT
SET name "Alice"
SET age 30
SET message [FORMAT "%s is %d years old" $name $age]
PUTS $message
```

8.5.2 String Cleaning

Cleaning User Input

```
PROC clean_input WITH text DO
  # Remove leading/trailing whitespace
  SET clean [STRING TRIM $text]

  # Convert to lowercase for consistency
  SET clean [STRING TOLOWER $clean]

  # Remove extra internal spaces
  WHILE [STRING FIRST "  " $clean] != -1 DO
    SET pos [STRING FIRST "  " $clean]
    SET clean [STRING REPLACE $clean $pos $pos+1 " "]
  END

  RETURN $clean
END

SET input "  HELLO    WORLD  "
SET clean [clean_input $input]
PUTS "Original: '$input'"
PUTS "Cleaned:  '$clean'"
```

Output:

```
Original: ’  HELLO    WORLD  ’
Cleaned:  ’hello world’
```

Tip

For complex string processing, consider using regular expressions (Chapter 9) for more powerful pattern matching and replacement.

BCL provides comprehensive file I/O capabilities.

Table 8.2: File Open Modes

Mode	Description
R	Read (file must exist)
W	Write (creates or truncates)
A	Append (creates if needed)
RW	Read and write

8.6 Opening and Closing Files

Basic File I/O

```
# Write to file
SET fh [OPEN "output.txt" W]
PUTS $fh "Line 1"
PUTS $fh "Line 2"
CLOSE $fh

# Read from file
SET fh [OPEN "output.txt" R]
SET content [READ $fh]
CLOSE $fh

PUTS "File content:"
PUTS $content
```

8.7 Reading Files Line by Line

Line-by-Line Reading

```
SET fh [OPEN "data.txt" R]

SET linenum 0
WHILE [EOF $fh] = 0 DO
  SET line [GETS $fh]
  INCR linenum

  IF [STRING LENGTH $line] > 0 THEN
    PUTS "Line $linenum: $line"
  END
END

CLOSE $fh
```

8.8 File Commands

FILE Commands

```
SET filename "test.txt"

# Check existence
IF [FILE EXISTS $filename] THEN
  PUTS "File exists"

  # Get size
  SET size [FILE SIZE $filename]
  PUTS "Size: $size bytes"

  # Rename
  FILE RENAME $filename "test_backup.txt"

  # Delete
  # FILE DELETE "test_backup.txt"
END

# Get current directory
SET cwd [PWD]
PUTS "Current directory: $cwd"

# Find files matching pattern
SET txtfiles [GLOB "*.txt"]
PUTS "Text files: $txtfiles"
```

Chapter 9

Regular Expressions

Regular expressions (often called "regex" or "regexp") are powerful patterns used to search, match, and manipulate text. Think of them as a sophisticated "find and replace" tool that can match complex patterns instead of just exact text.

9.1 What Are Regular Expressions?

Imagine you want to find all phone numbers in a document, or validate that an email address is correctly formatted, or replace all dates from one format to another. Regular expressions make these tasks easy.

Real-World Analogy

Regular expressions are like describing something without knowing its exact form:

- "Any word starting with 'cat'" - matches "cat", "cats", "category"
- "A sequence of digits" - matches "123", "4567", "999"
- "Text between quotes" - matches "hello", "goodbye"

Note

If you're new to programming, regular expressions might seem cryptic at first. Don't worry! We'll start with simple patterns and build up to more complex ones.

9.2 Basic Pattern Building Blocks

Regular expressions are built from simple pieces. Let's learn them one at a time.

9.2.1 Literal Characters

The simplest pattern is just normal text—it matches exactly what you write.

Literal Matching

```
SET text "The cat sat on the mat"

```

```
# Match the word "cat"
IF [REGEXP "cat" $text] THEN
  PUTS "Found 'cat' in the text"
END

# Match "mat"
IF [REGEXP "mat" $text] THEN
  PUTS "Found 'mat' in the text"
END

# This won't match because case matters
IF [REGEXP "Cat" $text] THEN
  PUTS "Found 'Cat'"
ELSE
  PUTS "'Cat' not found (wrong case)"
END
```

Output:

```
Found 'cat' in the text
Found 'mat' in the text
'Cat' not found (wrong case)
```

9.2.2 Special Characters - The Wildcards

Some characters have special meanings in regular expressions:

Table 9.1: Basic Regular Expression Characters

Symbol	Meaning
`.`	Matches any single character
`*`	Matches 0 or more of the previous character
`+`	Matches 1 or more of the previous character
`?`	Matches 0 or 1 of the previous character
`^`	Matches the start of the string
`$`	Matches the end of the string
`|`	OR operator (matches left or right)
`(...)`	Groups patterns together
`[...]`	Matches any character in the brackets
`\`	Escapes special characters

9.2.3 The Dot (.) - Any Character

The dot matches any single character except newline.

Using the Dot

```
# Match "c.t" - c, any character, then t
SET words [LIST "cat" "cot" "cut" "cart" "ct"]

```

```
FOREACH word IN $words DO
  IF [REGEXP "c.t" $word] THEN
    PUTS "$word matches c.t"
  ELSE
    PUTS "$word does NOT match c.t"
  END
END
```

Output:

```
cat matches c.t
cot matches c.t
cut matches c.t
cart matches c.t
ct does NOT match c.t
```

Explanation: The pattern `c.t` requires exactly one character between 'c' and 't'. "cart" matches because it contains "car" + "t" = "cart" which has the pattern.

9.2.4 Character Classes [...]

Square brackets match any one character from a set.

Character Classes

```
# Match c[aou]t - cat, cot, or cut
SET words [LIST "cat" "cot" "cut" "cit" "cet"]

FOREACH word IN $words DO
  IF [REGEXP "c\[aou\]t" $word] THEN
    PUTS "$word matches"
  END
END

# Match any digit [0-9]
SET text "I have 5 apples"
IF [REGEXP "\[0-9\]" $text] THEN
  PUTS "Contains a digit"
END

# Match any letter [a-z] or [A-Z]
IF [REGEXP "\[a-z\]" $text] THEN
  PUTS "Contains lowercase letters"
END
```

Output:

```
cat matches
cot matches
cut matches
Contains a digit
Contains lowercase letters
```

Tip

Common character classes:

- `[0-9]` - any digit
- `[a-z]` - any lowercase letter
- `[A-Z]` - any uppercase letter
- `[a-zA-Z]` - any letter
- `[^0-9]` - anything that's NOT a digit

9.2.5 Repetition: *, +, ?

These tell how many times a pattern should repeat.

Table 9.2: Repetition Operators

Operator	Meaning
`*`	0 or more times (optional, can repeat)
`+`	1 or more times (must appear at least once)
`?`	0 or 1 time (optional, appears once or not at all)
`{n}`	Exactly n times
`{n,}`	n or more times
`{n,m}`	Between n and m times

Repetition Examples

```
# Match one or more digits
SET texts [LIST "abc" "123" "abc123" "a1b2c3"]

FOREACH text IN $texts DO
  IF [REGEXP "\[0-9\]+" $text] THEN
    PUTS "$text contains numbers"
  END
END

# Match optional minus sign followed by digits: -?[0-9]+
SET numbers [LIST "123" "-456" "78" "-9"]
FOREACH num IN $numbers DO
  IF [REGEXP "^-?\[0-9\]+$" $num] THEN
    PUTS "$num is a valid number"
  END
END

# Match "color" or "colour"
IF [REGEXP "colou?r" "color"] THEN
  PUTS "Matches 'color'"
END
IF [REGEXP "colou?r" "colour"] THEN
```

```
  PUTS "Matches 'colour'"
END
```

Output:

```
123 contains numbers
abc123 contains numbers
a1b2c3 contains numbers
123 is a valid number
-456 is a valid number
78 is a valid number
-9 is a valid number
Matches 'color'
Matches 'colour'
```

9.2.6 Anchors: ^ and $

Anchors match positions, not characters.

- ^ matches the start of the string
- $ matches the end of the string

Using Anchors

```
SET text "hello world"

# Must start with "hello"
IF [REGEXP "^hello" $text] THEN
  PUTS "Starts with 'hello'"
END

# Must end with "world"
IF [REGEXP "world$" $text] THEN
  PUTS "Ends with 'world'"
END

# Must be EXACTLY "hello world" (nothing before or after)
IF [REGEXP "^hello world$" $text] THEN
  PUTS "Exact match"
END

# Won't match - has extra text
SET text2 "say hello world now"
IF [REGEXP "^hello world$" $text2] THEN
  PUTS "Exact match"
ELSE
  PUTS "Not an exact match - has extra text"
END
```

Output:

```
Starts with 'hello'
Ends with 'world'
```

```
Exact match
Not an exact match - has extra text
```

9.2.7 Shorthand Character Classes

BCL provides shortcuts for common patterns:

Table 9.3: Shorthand Character Classes

Shorthand	**Equivalent To**
`\d`	`[0-9]` - any digit
`\D`	`[^0-9]` - any non-digit
`\w`	`[a-zA-Z0-9_]` - word character
`\W`	Non-word character
`\s`	Whitespace (space, tab, newline)
`\S`	Non-whitespace

Shorthand Examples

```
# Find digits with \d
SET text "Room 101 is on floor 5"
IF [REGEXP "\\d+" $text] THEN
  PUTS "Found numbers"
END

# Find words with \w
IF [REGEXP "\\w+" $text] THEN
  PUTS "Found word characters"
END

# Validate format: word space word
SET input "hello world"
IF [REGEXP "^\\w+\\s+\\w+$" $input] THEN
  PUTS "Valid format: two words separated by space"
END
```

Output:

```
Found numbers
Found word characters
Valid format: two words separated by space
```

Warning

In BCL, you need to escape backslashes in strings. Write `\\d` to represent `\d` in the pattern.

9.3 The REGEXP Command

`REGEXP` checks if a pattern matches and can extract matched portions.

Syntax:

```
REGEXP pattern string                        # Returns 1 if match, 0 if not
REGEXP pattern string matchvar               # Store entire match
REGEXP pattern string matchvar subvar... # Store submatches
```

9.3.1 Basic Matching

Simple Pattern Matching

```
SET email "user@example.com"

# Check if it looks like an email
IF [REGEXP "@" $email] THEN
  PUTS "Contains @ symbol"
END

# Check for email pattern: word @ word . word
IF [REGEXP "\\w+@\\w+\\.\\w+" $email] THEN
  PUTS "Looks like a valid email"
END

# Validate phone number: exactly 10 digits
SET phone "5551234567"
IF [REGEXP "^\\d\{10\}$" $phone] THEN
  PUTS "Valid 10-digit phone number"
END
```

Output:

```
Contains @ symbol
Looks like a valid email
Valid 10-digit phone number
```

9.3.2 Capturing Matches

Use parentheses `(...)` to capture parts of the match.

Extracting Information

```
# Extract year from date
SET date "Today is 2025-10-22"
REGEXP "(\\d\{4\})-(\\d\{2\})-(\\d\{2\})" $date MATCH year month
    day
PUTS "Year: $year"
PUTS "Month: $month"
PUTS "Day: $day"

# Extract name and extension from filename
SET filename "document.pdf"
REGEXP "(.+)\\.(\\w+)$" $filename MATCH name ext
PUTS "Name: $name"
PUTS "Extension: $ext"

```

```
# Extract email parts
SET email "john.doe@example.com"
REGEXP "(.+)@(.+)" $email MATCH username domain
PUTS "Username: $username"
PUTS "Domain: $domain"
```

Output:

```
Year: 2025
Month: 10
Day: 22
Name: document
Extension: pdf
Username: john.doe
Domain: example.com
```

9.3.3 REGEXP Options

Table 9.4: REGEXP Options

Option	Description
`-nocase`	Case-insensitive matching
`-line`	Treat string as multiple lines (ˆ and $ match line starts/ends)
`-lineanchor`	Similar to -line
`-expanded`	Ignore whitespace in pattern (for readability)

Case-Insensitive Matching

```
SET text "Hello World"

# Case-sensitive (won't match)
IF [REGEXP "hello" $text] THEN
  PUTS "Found (sensitive)"
ELSE
  PUTS "Not found (sensitive)"
END

# Case-insensitive (will match)
IF [REGEXP -nocase "hello" $text] THEN
  PUTS "Found (insensitive)"
END
```

Output:

```
Not found (sensitive)
Found (insensitive)
```

9.4 The REGSUB Command

REGSUB replaces text that matches a pattern.

Syntax:

```
REGSUB pattern string replacement          # Replace first match
REGSUB pattern string replacement ALL      # Replace all matches
```

9.4.1 Basic Replacement

Simple Replacements

```
SET text "Hello, World!"

# Replace first occurrence
SET result [REGSUB "World" $text "BCL"]
PUTS $result  # Hello, BCL!

# Replace all occurrences
SET text2 "foo bar foo baz foo"
SET result2 [REGSUB "foo" $text2 "XXX" ALL]
PUTS $result2  # XXX bar XXX baz XXX

# Remove all digits
SET text3 "Room 101 is on floor 5"
SET clean [REGSUB "\\d+" $text3 "" ALL]
PUTS $clean  # Room  is on floor
```

Output:

```
Hello, BCL!
XXX bar XXX baz XXX
Room  is on floor
```

9.4.2 Using Captured Groups in Replacement

You can reference captured groups in the replacement using & or \1, \2, etc.

Advanced Replacements

```
# Swap first and last name
SET name "John Doe"
SET swapped [REGSUB "(\\w+) (\\w+)" $name "\\2, \\1"]
PUTS $swapped  # Doe, John

# Format phone number: 5551234567 -> (555) 123-4567
SET phone "5551234567"
SET formatted [REGSUB "(\\d\{3\})(\\d\{3\})(\\d\{4\})" $phone
   "(\\1) \\2-\\3"]
PUTS $formatted  # (555) 123-4567

# Add "http://" to URLs that don't have it
SET url "example.com"
IF [REGEXP "^http" $url] = 0 THEN
```

```
  SET url [REGSUB "^" $url "http://"]
END
PUTS $url  # http://example.com
```

Output:

```
Doe, John
(555) 123-4567
http://example.com
```

9.5 Practical Examples

9.5.1 Email Validator (Advanced)

Complete Email Validation

```
PROC validate_email WITH email DO
  # Basic pattern: user@domain.tld
  SET pattern "^\\w+(\\.\\w+)*@\\w+(\\.\\w+)+$"

  IF [REGEXP $pattern $email] THEN
    PUTS "$email is VALID"
    RETURN 1
  ELSE
    PUTS "$email is INVALID"
    RETURN 0
  END
END

# Test cases
validate_email "user@example.com"
validate_email "john.doe@company.co.uk"
validate_email "invalid@"
validate_email "@invalid.com"
validate_email "no-at-sign.com"
```

Output:

```
user@example.com is VALID
john.doe@company.co.uk is VALID
invalid@ is INVALID
@invalid.com is INVALID
no-at-sign.com is INVALID
```

9.5.2 Extract URLs from Text

Finding URLs

```
PROC extract_urls WITH text DO
  SET urls [LIST]

  # Pattern for http(s) URLs
  SET pattern "https?://\[\\w\\.-\]+\[/\\w\\.-\]*"

  SET pos 0
  WHILE 1 DO
    # Find next URL
    IF [REGEXP -start $pos $pattern $text url] THEN
      SET urls [LAPPEND $urls $url]
      # Move past this match
      SET pos [STRING FIRST $url $text $pos]
      INCR pos [STRING LENGTH $url]
    ELSE
      BREAK
    END
  END

  RETURN $urls
END

SET text "Visit http://example.com or https://bcl.org for info"
SET urls [extract_urls $text]
PUTS "Found URLs:"
FOREACH url IN $urls DO
  PUTS "  - $url"
END
```

Output:

```
Found URLs:
  - http://example.com
  - https://bcl.org
```

9.5.3 Password Strength Validator

Regex-based Password Checking

```
PROC check_password_strength WITH password DO
  SET score 0

  # Check length
  IF [STRING LENGTH $password] >= 8 THEN
    INCR score
  END

  # Check for lowercase
  IF [REGEXP "\[a-z\]" $password] THEN
    INCR score
  END

  # Check for uppercase
  IF [REGEXP "\[A-Z\]" $password] THEN
    INCR score
  END

  # Check for digits
  IF [REGEXP "\\d" $password] THEN
    INCR score
  END

  # Check for special characters
  IF [REGEXP "\[!@#$%^&*\]" $password] THEN
    INCR score
  END

  # Report strength
  IF $score < 3 THEN
    RETURN "Weak"
  ELSEIF $score < 5 THEN
    RETURN "Medium"
  ELSE
    RETURN "Strong"
  END
END

SET passwords [LIST "hello" "Hello123" "MyP@ss123" "Complex$Pass9"]
FOREACH pass IN $passwords DO
  SET strength [check_password_strength $pass]
  PUTS "$pass: $strength"
END
```

Output:

```
hello: Weak
Hello123: Medium
MyP@ss123: Strong
Complex$Pass9: Strong
```

9.5.4 Format Phone Numbers

Normalize Phone Numbers

```
PROC format_phone WITH phone DO
  # Remove all non-digits
  SET clean [REGSUB "\\D" $phone "" ALL]

  # Check if we have 10 digits
  IF [STRING LENGTH $clean] != 10 THEN
    RETURN "Invalid phone number"
  END

  # Format as (XXX) XXX-XXXX
  SET formatted [REGSUB "(\\d\{3\})(\\d\{3\})(\\d\{4\})" $clean
    "(\\1) \\2-\\3"]
  RETURN $formatted
END

# Test with various formats
SET phones [LIST "5551234567" "555-123-4567" "(555) 123-4567"
   "555.123.4567"]
FOREACH phone IN $phones DO
  SET formatted [format_phone $phone]
  PUTS "$phone -> $formatted"
END
```

Output:

```
5551234567 -> (555) 123-4567
555-123-4567 -> (555) 123-4567
(555) 123-4567 -> (555) 123-4567
555.123.4567 -> (555) 123-4567
```

9.6 Common Regular Expression Patterns

Here's a reference of useful patterns for common tasks:

Table 9.5: Common Regex Patterns

Pattern	Description
`^\d{4}-\d{2}-\d{2}$`	Date (YYYY-MM-DD)
`\w+@\w+\.\w+`	Simple email
`^\d{3}-\d{3}-\d{4}$`	Phone (XXX-XXX-XXXX)
`^[01]?\d\d?$`	Number 0-199
`\b\w{3}\b`	Exactly 3-letter word
`https?://.*`	HTTP or HTTPS URL
`^\s*$`	Empty or whitespace only
`\d+\.\d{2}`	Decimal with 2 places

Tip

Regular expressions can get complex. Start simple and build up gradually. Test your patterns with various inputs to ensure they work as expected.

Warning

Be careful with patterns like `.*` (match anything) as they can match more than you expect. Use specific patterns when possible.

The **`CLOCK`** command provides time and date functionality.

Time Operations

```
# Get current time
SET now [CLOCK SECONDS]
PUTS "Timestamp: $now"

# Format timestamp
SET formatted [CLOCK FORMAT $now FORMAT "%Y-%m-%d %H:%M:%S"]
PUTS "Date: $formatted"

# Parse date string
SET timestamp [CLOCK SCAN "2025-12-25 00:00:00"]
PUTS "Christmas: $timestamp"

# Add time
SET tomorrow [CLOCK ADD $now 1 day]
SET next_week [CLOCK ADD $now 7 days]
PUTS "Tomorrow: [CLOCK FORMAT $tomorrow]"
```

Chapter 10

System Interaction

BCL provides several commands to interact with the operating system, execute code dynamically, load external files, and control program flow. This chapter covers these powerful system-level features.

10.1 Dynamic Code Execution

10.1.1 EVAL - Execute String as Code

`EVAL` evaluates a string as BCL code, allowing you to run code that's constructed at runtime.

Syntax:

```
EVAL string
```

Basic EVAL Usage

```
# Execute code from a string
SET command "PUTS 'Hello from EVAL'"
EVAL $command

# Build code dynamically
SET varname "result"
SET value 42
SET code "SET $varname $value"
EVAL $code
PUTS "result = $result"   # Prints: result = 42

# Evaluate expressions
SET expr "5 + 3 * 2"
SET answer [EVAL "EXPR $expr"]
PUTS "Answer: $answer"   # Prints: Answer: 11
```

Output:

```
Hello from EVAL
result = 42
Answer: 11
```

Warning

EVAL executes code in the current scope with access to all variables. Be extremely careful when evaluating user input, as it can execute arbitrary code. Never use **EVAL** with untrusted data!

10.1.2 Practical EVAL Examples

Dynamic Command Dispatcher

```
PROC dispatch_command WITH cmd DO
  # Map user commands to BCL code
  SWITCH $cmd
    CASE "greet" DO
      EVAL "PUTS 'Hello, user!'"
    END
    CASE "time" DO
      EVAL "PUTS [CLOCK FORMAT [CLOCK SECONDS]]"
    END
    CASE "random" DO
      EVAL "PUTS [EXPR rand() * 100]"
    END
    DEFAULT DO
      PUTS "Unknown command: $cmd"
    END
  END
END

dispatch_command "greet"
dispatch_command "time"
dispatch_command "random"
```

Simple Calculator with EVAL

```
PROC calculate WITH expression DO
  # Add EXPR wrapper and evaluate
  SET code "EXPR $expression"

  # Use error handling (if available)
  SET result [EVAL $code]

  RETURN $result
END

PUTS "5 + 3 = [calculate "5 + 3"]"
PUTS "10 * 2.5 = [calculate "10 * 2.5"]"
PUTS "sqrt(16) = [calculate "sqrt(16)"]"
```

Output:

```
5 + 3 = 8
10 * 2.5 = 25.0
sqrt(16) = 4.0
```

10.2 Loading External Code

10.2.1 SOURCE - Load and Execute Files

SOURCE loads and executes BCL code from an external file, allowing you to organize large programs into modules.

Syntax:

```
SOURCE filepath
```

Using SOURCE

Create a file `library.bcl`:

```
# library.bcl - Math utilities
PROC square WITH x DO
  RETURN [EXPR $x * $x]
END

PROC cube WITH x DO
  RETURN [EXPR $x * $x * $x]
END

SET PI 3.14159
```

Main script:

```
# Load the library
SOURCE "library.bcl"

# Use functions and variables from library
PUTS "5 squared = [square 5]"
PUTS "3 cubed = [cube 3]"
PUTS "PI = $PI"
```

Output:

```
5 squared = 25
3 cubed = 27
PI = 3.14159
```

Tip

Use **SOURCE** to:

- Split large programs into manageable files
- Create reusable library modules
- Share common code between multiple scripts
- Keep configuration in separate files

10.2.2 Practical SOURCE Examples

Configuration File

Create `config.bcl`:

```
# Application configuration
SET APP_NAME "MyApp"
SET APP_VERSION "1.0.0"
SET DEBUG_MODE 1
SET MAX_USERS 100
```

Main application:

```
# Load configuration
SOURCE "config.bcl"

PUTS "=== $APP_NAME v$APP_VERSION ==="
PUTS "Debug mode: $DEBUG_MODE"
PUTS "Max users: $MAX_USERS"

IF $DEBUG_MODE THEN
  PUTS "[DEBUG] Configuration loaded successfully"
END
```

Output:

```
=== MyApp v1.0.0 ===
Debug mode: 1
Max users: 100
[DEBUG] Configuration loaded successfully
```

Module System

Create `string_utils.bcl`:

```
PROC reverse_string WITH str DO
  RETURN [STRING REVERSE $str]
END

PROC count_vowels WITH str DO
  SET count 0
  SET len [STRING LENGTH $str]
  FOR 0 TO $len-1 DO
    SET char [STRING INDEX $str $__FOR]
    IF [REGEXP -nocase "\[aeiou\]" $char] THEN
      INCR count
    END
  END
  RETURN $count
END
```

Main script:

```
SOURCE "string_utils.bcl"

SET text "Hello World"
```

```
PUTS "Original: $text"
PUTS "Reversed: [reverse_string $text]"
PUTS "Vowels: [count_vowels $text]"
```

Output:

```
Original: Hello World
Reversed: dlroW olleH
Vowels: 3
```

10.3 Timing and Delays

10.3.1 AFTER - Pause Execution

`AFTER` suspends execution for a specified number of milliseconds.

Syntax:

```
AFTER milliseconds
```

Using AFTER

```
PUTS "Starting..."
AFTER 1000         # Wait 1 second (1000 ms)
PUTS "1 second later"

AFTER 500          # Wait 0.5 seconds
PUTS "0.5 seconds later"

AFTER 2000         # Wait 2 seconds
PUTS "Done!"
```

Output: (with delays)

```
Starting...
[1 second pause]
1 second later
[0.5 second pause]
0.5 seconds later
[2 second pause]
Done!
```

Progress Animation

```
PROC show_progress WITH steps DO
  PUTSN "Progress: "

  FOR 1 TO $steps DO
    PUTSN "."
    AFTER 200
  END

  PUTS " Done!"
```

```
END

PUTS "Loading"
show_progress 10
PUTS "Complete!"
```

Output:

```
Loading
Progress: .......... Done!
Complete!
```

Countdown Timer

```
PROC countdown WITH seconds DO
  FOR $seconds TO 1 STEP -1 DO
    PUTS "Time remaining: $__FOR seconds"
    AFTER 1000
  END
  PUTS "Time's up!"
END

countdown 5
```

Output:

```
Time remaining: 5 seconds
Time remaining: 4 seconds
Time remaining: 3 seconds
Time remaining: 2 seconds
Time remaining: 1 seconds
Time's up!
```

10.4 System Command Execution

10.4.1 EXEC - Execute System Commands

`EXEC` executes an operating system command and returns its output.

Syntax:

```
EXEC command [arg1 arg2 ...]
```

Warning

`EXEC` is typically only available on PC/desktop systems, not on embedded microcontrollers. It also poses security risks if used with untrusted input.

Basic EXEC Usage

```
# List files (Linux/Unix)
SET files [EXEC "ls"]
```

```
PUTS "Files:\n$files"

# Get current date
SET date [EXEC "date"]
PUTS "Current date: $date"

# Echo text
SET output [EXEC "echo" "Hello from shell"]
PUTS $output

# Count files
SET count [EXEC "ls" "-1" "|" "wc" "-l"]
PUTS "File count: $count"
```

Practical EXEC Examples

```
# Check if a file exists (using shell command)
PROC file_exists_shell WITH filename DO
  # Use test command
  SET result [EXEC "test" "-f" $filename "&&" "echo" "1" "||"
      "echo" "0"]
  RETURN [STRING TRIM $result]
END

# Get disk usage
PROC get_disk_usage DO
  SET usage [EXEC "df" "-h" "."]
  RETURN $usage
END

# Backup a file
PROC backup_file WITH filename DO
  SET timestamp [CLOCK FORMAT [CLOCK SECONDS] FORMAT
      "%Y%m%d_%H%M%S"]
  SET backup_name "$filename.backup_$timestamp"
  EXEC "cp" $filename $backup_name
  PUTS "Backed up to: $backup_name"
END

backup_file "important.txt"
```

10.5 Environment Variables

10.5.1 ENV - Access Environment Variables

`ENV` reads environment variables from the operating system.

Syntax:

```
ENV variable_name
```

Reading Environment Variables

```
# Get home directory
SET home [ENV HOME]
PUTS "Home directory: $home"

# Get current user
SET user [ENV USER]
PUTS "Current user: $user"

# Get PATH
SET path [ENV PATH]
PUTS "PATH: $path"

# Check if a variable exists
SET editor [ENV EDITOR]
IF [STRING LENGTH $editor] > 0 THEN
  PUTS "Default editor: $editor"
ELSE
  PUTS "No EDITOR set, using vi"
  SET editor "vi"
END
```

Output:

```
Home directory: /home/user
Current user: user
PATH: /usr/local/bin:/usr/bin:/bin
Default editor: vim
```

Cross-Platform Paths

```
PROC get_temp_dir DO
  # Try different environment variables
  SET temp [ENV TMPDIR]
  IF [STRING LENGTH $temp] = 0 THEN
    SET temp [ENV TEMP]
  END
  IF [STRING LENGTH $temp] = 0 THEN
    SET temp [ENV TMP]
  END
  IF [STRING LENGTH $temp] = 0 THEN
    SET temp "/tmp"  # Default for Unix
  END
  RETURN $temp
END

SET tempdir [get_temp_dir]
PUTS "Temp directory: $tempdir"
```

10.6 Command Line Arguments

10.6.1 ARGV - Get Script Arguments

ARGV returns the list of command-line arguments passed to the script.

Syntax:

```
ARGV
```

Processing Arguments

Save as `args.bcl`:

```
# Get arguments
SET args [ARGV]
SET count [LLENGTH $args]

PUTS "Received $count argument(s):"

IF $count == 0 THEN
  PUTS "  (none)"
ELSE
  SET i 1
  FOREACH arg IN $args DO
    PUTS "  $i. $arg"
    INCR i
  END
END
```

Run as: `bcl args.bcl one two three`

Output:

```
Received 3 argument(s):
  1. one
  2. two
  3. three
```

Command-Line Tool

```
# greet.bcl - A simple greeting tool
SET args [ARGV]
SET argc [LLENGTH $args]

IF $argc == 0 THEN
  PUTS "Usage: bcl greet.bcl <name> [title]"
  EXIT 1
END

SET name [LINDEX $args 0]

IF $argc >= 2 THEN
  SET title [LINDEX $args 1]
  PUTS "Hello, $title $name!"
ELSE
  PUTS "Hello, $name!"
```

```
END
```

Usage:

```
$ bcl greet.bcl Alice
Hello, Alice!

$ bcl greet.bcl Bob Dr.
Hello, Dr. Bob!
```

10.7 Program Termination

10.7.1 EXIT - Terminate Execution

`EXIT` terminates the BCL script or interpreter and returns an exit code to the operating system.

Syntax:

```
EXIT [code]
```

The `code` is optional:

- `0` indicates success (default)
- Non-zero indicates an error or specific condition

Using EXIT

```
SET args [ARGV]

IF [LLENGTH $args] == 0 THEN
  PUTS "Error: No arguments provided"
  EXIT 1     # Exit with error code
END

# Process arguments...
PUTS "Processing..."

# Success
EXIT 0
```

Exit Codes for Different Errors

```
PROC validate_file WITH filename DO
  # Check if file exists
  IF [FILE EXISTS $filename] = 0 THEN
    PUTS "Error: File '$filename' not found"
    EXIT 2     # File not found
  END

  # Check if file is readable
  # (assuming FILE READABLE command exists)
  # ...

  PUTS "File validated successfully"
END

SET args [ARGV]
IF [LLENGTH $args] == 0 THEN
  PUTS "Usage: script.bcl <filename>"
  EXIT 1     # Invalid usage
END

validate_file [LINDEX $args 0]
PUTS "Done!"
EXIT 0       # Success
```

10.8 Practical System Integration Examples

10.8.1 Script Launcher

Dynamic Script Loader

```
PROC load_plugin WITH name DO
  SET plugin_file "plugins/$name.bcl"

  IF [FILE EXISTS $plugin_file] THEN
    PUTS "Loading plugin: $name"
    SOURCE $plugin_file
    RETURN 1
  ELSE
    PUTS "Plugin not found: $name"
    RETURN 0
  END
END

# Load multiple plugins
SET plugins [LIST "database" "network" "utils"]
FOREACH plugin IN $plugins DO
  load_plugin $plugin
END
```

10.8.2 Interactive Shell

Simple BCL Shell

```
PUTS "=== BCL Interactive Shell ==="
PUTS "Type 'exit' or 'quit' to exit"
PUTS ""

WHILE 1 DO
  PUTSN "BCL> "
  SET input [GETS stdin]

  # Trim input
  SET input [STRING TRIM $input]

  # Check for exit
  IF $input == "exit" OR $input == "quit" THEN
    PUTS "Goodbye!"
    BREAK
  END

  # Skip empty lines
  IF [STRING LENGTH $input] == 0 THEN
    CONTINUE
  END

  # Evaluate input as BCL code
  EVAL $input
END
```

10.8.3 Build Script

Automated Build System

```
PROC run_command WITH description command DO
  PUTS ">>> $description"
  SET result [EXEC $command]
  PUTS $result
  RETURN [STRING LENGTH $result]
END

PROC build DO
  PUTS "==================================="
  PUTS "  Starting Build Process"
  PUTS "==================================="
  PUTS ""

  # Clean
  run_command "Cleaning old files..." "make clean"
  AFTER 500

  # Build
  run_command "Compiling..." "make"
  AFTER 500

  # Test
  run_command "Running tests..." "make test"
  AFTER 500

  PUTS ""
  PUTS "==================================="
  PUTS "  Build Complete!"
  PUTS "==================================="
END
```

```

# Check arguments
SET args [ARGV]
IF [LLENGTH $args] > 0 THEN
  SET target [LINDEX $args 0]
  IF $target == "build" THEN
    build
  ELSEIF $target == "clean" THEN
    EXEC "make clean"
  ELSE
    PUTS "Unknown target: $target"
    EXIT 1
  END
ELSE
  PUTS "Usage: build.bcl <build|clean>"
  EXIT 1
END
```

10.9 System Commands Summary

Table 10.1: System Interaction Commands

Command	Description
`EVAL string`	Execute string as BCL code
`SOURCE file`	Load and execute external BCL file
`AFTER ms`	Pause execution for milliseconds
`EXEC cmd`	Execute system command, return output
`ENV name`	Get environment variable
`ARGV`	Get command-line arguments list
`EXIT [code]`	Terminate program with exit code

Warning

Security Considerations:

- Never use **`EVAL`** or **`EXEC`** with untrusted user input
- Validate and sanitize all external data before using it in commands
- Be careful with **`SOURCE`** to avoid loading malicious code
- Use exit codes consistently for better shell script integration

Tip

Best Practices:

- Use **`SOURCE`** to organize large projects into modules
- Store configuration in separate files loaded with **`SOURCE`**

- Use environment variables for system-specific settings
- Provide helpful error messages and appropriate exit codes
- Document your modules and libraries clearly

Chapter 11

Introspection

Introspection is the ability of a program to examine its own structure and state while running. BCL provides the **INFO** command, which lets you inspect variables, procedures, and the BCL environment itself.

11.1 What is Introspection?

Think of introspection as your program's ability to "look in the mirror" and see what it contains. It can ask questions like:

- "Does a variable named 'x' exist?"
- "What procedures have I defined?"
- "What are the parameters of this procedure?"
- "What version of BCL am I running?"

This is incredibly useful for:

- **Debugging** - Check if variables exist before using them
- **Dynamic code** - Make decisions based on what's available
- **Error handling** - Verify preconditions before operations
- **Development tools** - Build debuggers, profilers, or IDEs

11.2 The INFO Command

The `INFO` command is your gateway to introspection. It has many subcommands, each providing different information about your program's state.

General Syntax:

```
INFO subcommand [arguments...]
```

11.3 Checking Variables

11.3.1 INFO EXISTS - Does a Variable Exist?

`INFO EXISTS` checks if a variable has been created and assigned a value.

Syntax:

```
INFO EXISTS varname
```

Returns 1 if the variable exists, 0 if it doesn't.

Checking Variable Existence

```
SET username "Alice"

# Check if variable exists
IF [INFO EXISTS username] THEN
  PUTS "username exists: $username"
ELSE
  PUTS "username doesn't exist"
END

# Check non-existent variable
IF [INFO EXISTS password] THEN
  PUTS "password exists"
ELSE
  PUTS "password doesn't exist - need to set it"
  SET password "secret123"
END

# Now it exists
IF [INFO EXISTS password] THEN
  PUTS "password now exists"
END
```

Output:

```
username exists: Alice
password doesn't exist - need to set it
password now exists
```

Tip

Use `INFO EXISTS` to avoid errors when accessing optional variables or user input that might not have been provided.

11.3.2 INFO VARS - List All Variables

`INFO VARS` returns a list of all variables in the current scope.

Syntax:

```
INFO VARS                # All variables
INFO VARS pattern        # Variables matching pattern
```

Listing Variables

```
SET name "Alice"
SET age 30
SET city "New York"
SET temp_value 123

# List all variables
SET all_vars [INFO VARS]
PUTS "All variables: $all_vars"

# List variables matching pattern
SET temp_vars [INFO VARS "temp_*"]
PUTS "Temp variables: $temp_vars"

# List variables starting with specific letter
SET a_vars [INFO VARS "a*"]
PUTS "Variables starting with 'a': $a_vars"
```

Output:

```
All variables: name age city temp_value
Temp variables: temp_value
Variables starting with 'a': age
```

11.3.3 INFO GLOBALS - List Global Variables

`INFO GLOBALS` returns a list of all global variables.

Global Variables

```
SET global_config "production"
SET global_debug 1

PROC test DO
  SET local_var "I'm local"

  # From inside proc, check globals
  SET globals [INFO GLOBALS]
  PUTS "Global variables: $globals"

  # Check locals
  SET locals [INFO LOCALS]
  PUTS "Local variables: $locals"
END

test
```

Output:

```
Global variables: global_config global_debug
Local variables: local_var
```

11.3.4 INFO LOCALS - List Local Variables

`INFO LOCALS` returns variables local to the current procedure.

Local Variable Inspection

```
PROC calculate WITH x y DO
  SET sum [EXPR $x + $y]
  SET product [EXPR $x * $y]
  SET temp "working..."

  # Show all local variables (including parameters)
  SET locals [INFO LOCALS]
  PUTS "Local variables in procedure: $locals"

  # Show just local temp variables
  SET temps [INFO LOCALS "temp*"]
  PUTS "Temp variables: $temps"
END

calculate 5 3
```

Output:

```
Local variables in procedure: x y sum product temp
Temp variables: temp
```

11.4 Inspecting Procedures

11.4.1 INFO PROCS - List All Procedures

`INFO PROCS` returns a list of all defined procedures.

Syntax:

```
INFO PROCS                # All procedures
INFO PROCS pattern        # Procedures matching pattern
```

Listing Procedures

```
PROC calculate WITH x y DO
  RETURN [EXPR $x + $y]
END

PROC format_output WITH text DO
  PUTS "=== $text ==="
END

PROC helper_function DO
  # Helper code
END

# List all procedures
SET procs [INFO PROCS]
PUTS "All procedures:"
```

```
FOREACH proc IN $procs DO
  PUTS "  - $proc"
END

# List only helper procedures
SET helpers [INFO PROCS "helper_*"]
PUTS "\nHelper procedures: $helpers"
```

Output:

```
All procedures:
  - calculate
  - format_output
  - helper_function

Helper procedures: helper_function
```

11.4.2 INFO ARGS - Get Procedure Parameters

`INFO ARGS` returns the parameter names for a procedure.

Syntax:

```
INFO ARGS procname
```

Inspecting Procedure Parameters

```
PROC greet WITH name @title DO
  IF [INFO EXISTS title] THEN
    PUTS "Hello, $title $name"
  ELSE
    PUTS "Hello, $name"
  END
END

# Get parameter list
SET params [INFO ARGS greet]
PUTS "Parameters of 'greet': $params"

# Check how many parameters
SET count [LLENGTH $params]
PUTS "Number of parameters: $count"
```

Output:

```
Parameters of 'greet': name title
Number of parameters: 2
```

11.4.3 INFO BODY - Get Procedure Body

`INFO BODY` returns the actual code (body) of a procedure.

Syntax:

```
INFO BODY procname
```

Examining Procedure Code

```
PROC add WITH a b DO
  SET result [EXPR $a + $b]
  RETURN $result
END

# Get the procedure body
SET body [INFO BODY add]
PUTS "Procedure 'add' contains:"
PUTS $body
```

Output:

```
Procedure 'add' contains:
  SET result [EXPR $a + $b]
  RETURN $result
```

Tip

`INFO BODY` is useful for debugging, creating documentation, or implementing code analysis tools.

11.5 System Information

11.5.1 INFO BCLVERSION - Get BCL Version

`INFO BCLVERSION` returns the version of BCL you're running.

Version Check

```
SET version [INFO BCLVERSION]
PUTS "Running BCL version: $version"

# Version-specific features
IF [STRING MATCH "1.5*" $version] THEN
  PUTS "You have BCL 1.5.x - all features available"
ELSE
  PUTS "Consider upgrading to BCL 1.5 or newer"
END
```

Output:

```
Running BCL version: 1.5.1
You have BCL 1.5.x - all features available
```

11.5.2 INFO COMMANDS - List All Available Commands

`INFO COMMANDS` returns a list of all BCL commands available in the current interpreter.

Available Commands

```
# Get all commands
SET commands [INFO COMMANDS]
PUTS "Total commands available: [LLENGTH $commands]"

# Check if a specific command exists
IF [LSEARCH $commands "REGEXP"] != -1 THEN
  PUTS "REGEXP command is available"
END

# List string-related commands
SET string_cmds [LIST]
FOREACH cmd IN $commands DO
  IF [STRING MATCH "STRING*" $cmd] THEN
    SET string_cmds [LAPPEND $string_cmds $cmd]
  END
END
PUTS "String commands: $string_cmds"
```

Output:

```
Total commands available: 87
REGEXP command is available
String commands: STRING
```

11.6 Practical Applications

11.6.1 Safe Variable Access

Avoid Errors with INFO EXISTS

```
PROC safe_print WITH varname DO
  # Check if variable exists before accessing
  IF [INFO EXISTS $varname] THEN
    # Get the value using SET without argument
    SET value [SET $varname]
    PUTS "$varname == $value"
  ELSE
    PUTS "Variable '$varname' not found"
  END
END

SET username "Alice"

safe_print username
safe_print password
safe_print email
```

Output:

```
username = Alice
Variable 'password' not found
Variable 'email' not found
```

11.6.2 Dynamic Configuration

Optional Configuration Variables

```
PROC load_config DO
  # Set defaults
  SET config_host "localhost"
  SET config_port 8080
  SET config_debug 0

  # Check if user provided custom values
  IF [INFO EXISTS USER_HOST] THEN
    SET config_host $USER_HOST
    PUTS "Using custom host: $config_host"
  END

  IF [INFO EXISTS USER_PORT] THEN
    SET config_port $USER_PORT
    PUTS "Using custom port: $config_port"
  END

  IF [INFO EXISTS USER_DEBUG] THEN
    SET config_debug $USER_DEBUG
    PUTS "Debug mode: $config_debug"
  END

  RETURN [LIST $config_host $config_port $config_debug]
END

# Load with defaults
SET config [load_config]
PUTS "Config: $config"

# Now set custom values
SET USER_HOST "192.168.1.100"
SET USER_PORT 3000
SET config [load_config]
PUTS "Custom config: $config"
```

Output:

```
Config: localhost 8080 0
Using custom host: 192.168.1.100
Using custom port: 3000
Custom config: 192.168.1.100 3000 0
```

11.6.3 Help System

Interactive Help

```
PROC show_help WITH @command DO
  IF [INFO EXISTS command] THEN
    # Show help for specific command
    IF [LSEARCH [INFO PROCS] $command] != -1 THEN
```

```
      PUTS "Procedure: $command"
      SET params [INFO ARGS $command]
      PUTS "Parameters: $params"
      PUTS ""
      PUTS "Body:"
      PUTS [INFO BODY $command]
    ELSE
      PUTS "Command '$command' not found"
    END
  ELSE
    # Show all available procedures
    PUTS "Available procedures:"
    SET procs [INFO PROCS]
    FOREACH proc IN $procs DO
      SET params [INFO ARGS $proc]
      PUTS "  $proc ($params)"
    END
  END
END

PROC calculate_area WITH width height DO
  RETURN [EXPR $width * $height]
END

# Show all procedures
show_help

# Show help for specific procedure
PUTS ""
show_help calculate_area
```

Output:

```
Available procedures:
  show_help (command)
  calculate_area (width height)

Procedure: calculate_area
Parameters: width height

Body:
  RETURN [EXPR $width * $height]
```

11.6.4 Debugging Tool

Variable Dumper

```
PROC dump_variables WITH @pattern DO
  # Default pattern: all variables
  IF [INFO EXISTS pattern] = 0 THEN
    SET pattern "*"
  END

  SET vars [INFO VARS $pattern]

  IF [LLENGTH $vars] = 0 THEN
    PUTS "No variables match pattern '$pattern'"
    RETURN
  END

  PUTS "=== Variables matching '$pattern' ==="
  FOREACH var IN $vars DO
    SET value [SET $var]
    SET type "string"

    # Try to determine type
    IF [REGEXP "^-?\[0-9\]+$" $value] THEN
      SET type "integer"
    ELSEIF [REGEXP "^-?\[0-9\]+\\.\[0-9\]+$" $value] THEN
      SET type "float"
    END

    PUTS [FORMAT "  %-15s = %-20s (%s)" $var $value $type]
  END
END

# Create some test variables
SET name "Alice"
SET age 30
SET height 1.68
SET count 42
SET debug_flag 1

# Dump all variables
dump_variables

# Dump only specific pattern
PUTS ""
dump_variables "debug_*"
```

Output:

```
=== Variables matching '*' ===
  name            = Alice                (string)
  age             = 30                   (integer)
  height          = 1.68                 (float)
  count           = 42                   (integer)
  debug_flag      = 1                    (integer)

=== Variables matching 'debug_*' ===
  debug_flag      = 1                    (integer)
```

11.6.5 Procedure Documentation Generator

Auto-Generate Documentation

```
PROC document_procedures DO
  SET procs [INFO PROCS]

  PUTS "===================================="
  PUTS "  BCL Procedure Documentation"
  PUTS "===================================="
  PUTS ""

  FOREACH proc IN $procs DO
    # Skip internal procedures
    IF [STRING MATCH "_*" $proc] THEN
      CONTINUE
    END

    SET params [INFO ARGS $proc]
    SET param_count [LLENGTH $params]

    PUTS "PROCEDURE: $proc"
    PUTS "Parameters: $param_count"

    IF $param_count > 0 THEN
      SET i 1
      FOREACH param IN $params DO
        PUTS "  $i. $param"
        INCR i
      END
    END

    PUTS ""
  END
END

# Define some example procedures
PROC add WITH a b DO
  RETURN [EXPR $a + $b]
END

PROC greet WITH name @title DO
  PUTS "Hello, $name"
END

PROC _internal_helper DO
  # This won't be documented
END

# Generate documentation
document_procedures
```

Output:

```
====================================
  BCL Procedure Documentation
```

```
===================================

PROCEDURE: document_procedures
Parameters: 0

PROCEDURE: add
Parameters: 2
  1. a
  2. b

PROCEDURE: greet
Parameters: 2
  1. name
  2. title
```

11.7 INFO Command Reference

Table 11.1: INFO Subcommands Reference

Subcommand	Description
`EXISTS varname`	Check if variable exists
`VARS [pattern]`	List all variables (or matching pattern)
`GLOBALS [pattern]`	List global variables
`LOCALS [pattern]`	List local variables
`PROCS [pattern]`	List all procedures
`COMMANDS [pattern]`	List all BCL commands
`ARGS procname`	Get procedure parameters
`BODY procname`	Get procedure body (code)
`BCLVERSION`	Get BCL version string

Tip

Introspection is powerful for building development tools, debuggers, and self-documenting code. Use it to make your programs smarter and more robust!

Warning

Be careful when using `INFO BODY` with user-provided procedure names, as it exposes your code. In production systems, consider restricting access to introspection commands.

11.8 Text File Processor

Word Counter

```
PROC count_words WITH filename DO
  IF [FILE EXISTS $filename] = 0 THEN
    PUTS "Error: File not found"
    RETURN 0
  END

  SET fh [OPEN $filename R]
  SET content [READ $fh]
  CLOSE $fh

  # Split on whitespace
  SET words [SPLIT $content " "]
  SET count [LLENGTH $words]

  RETURN $count
END

SET wc [count_words "document.txt"]
PUTS "Word count: $wc"
```

11.9 Simple Calculator

Interactive Calculator

```
PUTS "Simple Calculator"
PUTS "Enter expressions (e.g., 5 + 3)"
PUTS "Type 'quit' to exit"
PUTS ""

SET running 1
WHILE $running DO
  PUTSN "> "
  SET input [GETS stdin]

  IF $input == "quit" THEN
    SET running 0
    CONTINUE
  END

  SET result [EXPR $input]
  PUTS "= $result"
END

PUTS "Goodbye!"
```

11.10 Todo List Manager

Todo List Application

```
SET todos [LIST]

PROC add_todo WITH task DO
  GLOBAL todos
  SET todos [LAPPEND $todos $task]
  PUTS "Added: $task"
END

PROC list_todos DO
  GLOBAL todos
  SET count [LLENGTH $todos]

  IF $count == 0 THEN
    PUTS "No todos"
    RETURN
  END

  PUTS "Todo List:"
  SET i 0
  FOREACH todo IN $todos DO
    INCR i
    PUTS "  $i. $todo"
  END
END

PROC remove_todo WITH index DO
  GLOBAL todos
  SET idx [EXPR $index - 1]
  SET count [LLENGTH $todos]

  IF $idx >= 0 AND $idx < $count THEN
    SET todos [LREPLACE $todos $idx $idx]
    PUTS "Removed todo #$index"
  ELSE
    PUTS "Invalid index"
  END
END

# Main loop
SET running 1
WHILE $running DO
  PUTS ""
  PUTS "1. Add todo"
  PUTS "2. List todos"
  PUTS "3. Remove todo"
  PUTS "4. Quit"
  PUTSN "Choice: "

  SET choice [GETS stdin]

  SWITCH $choice DO
```

```
    CASE "1"
      PUTS "Enter task:"
      SET task [GETS stdin]
      add_todo $task
    CASE "2"
      list_todos
    CASE "3"
      PUTS "Enter todo number:"
      SET num [GETS stdin]
      remove_todo $num
    CASE "4"
      SET running 0
    DEFAULT
      PUTS "Invalid choice"
  END
END
```

11.11 Log File Analyzer

Analyze Logs

```
PROC analyze_log WITH logfile DO
  IF [FILE EXISTS $logfile] = 0 THEN
    PUTS "Log file not found"
    RETURN
  END

  SET fh [OPEN $logfile R]

  SET errors 0
  SET warnings 0
  SET info 0
  SET total 0

  WHILE [EOF $fh] = 0 DO
    SET line [GETS $fh]
    INCR total

    IF [REGEXP "ERROR" $line NOCASE] THEN
      INCR errors
    ELSEIF [REGEXP "WARN" $line NOCASE] THEN
      INCR warnings
    ELSEIF [REGEXP "INFO" $line NOCASE] THEN
      INCR info
    END
  END

  CLOSE $fh

  PUTS "Log Analysis Results:"
  PUTS "  Total lines: $total"
  PUTS "  Errors: $errors"
  PUTS "  Warnings: $warnings"
  PUTS "  Info: $info"
END

analyze_log "application.log"
```

This chapter provides a quick alphabetical reference of all BCL commands.

Table 11.2: BCL Command Reference

Command	Description
`AFTER`	Pause execution for milliseconds
`APPEND`	Append text to variable
`ARGV`	Get command-line arguments
`BREAK`	Exit current loop
`CLOCK`	Time and date operations
`CLOSE`	Close file handle
`CONCAT`	Concatenate lists
`CONTINUE`	Skip to next loop iteration
`ENV`	Access environment variables
`EOF`	Check end of file
`EVAL`	Evaluate string as code
`EXEC`	Execute system command
`EXIT`	Terminate program
`EXPR`	Evaluate expression
`FILE`	File operations (EXISTS, SIZE, DELETE, RENAME)
`FOR`	For loop
`FOREACH`	Iterate over list
`FORMAT`	Format string (printf-style)
`GETS`	Read line from input
`GLOB`	File pattern matching
`GLOBAL`	Declare global variable
`IF`	Conditional execution
`INCR`	Increment variable
`INFO`	Introspection commands
`JOIN`	Join list to string
`LAPPEND`	Append to list
`LINDEX`	Get list element
`LINSERT`	Insert into list
`LIST`	Create list
`LLENGTH`	Get list length
`LRANGE`	Extract sublist
`LREPLACE`	Replace list elements
`LSEARCH`	Search list
`LSORT`	Sort list
`OPEN`	Open file
`PROC`	Define procedure
`PUTS`	Print with newline

Table 11.2 – continued

Command	**Description**
`PUTSN`	Print without newline
`PWD`	Get current directory
`READ`	Read from file
`REGEXP`	Regular expression matching
`REGSUB`	Regular expression substitution
`RETURN`	Return from procedure
`SCAN`	Parse formatted string
`SEEK`	Set file position
`SET`	Assign/read variable
`SOURCE`	Execute script file
`SPLIT`	Split string to list
`STRING`	String operations
`SWITCH`	Multi-way branch
`TELL`	Get file position
`UNSET`	Delete variable
`WHILE`	While loop

Chapter 12

Associative Arrays

BCL supports associative arrays (also known as dictionaries or hash maps in other languages), inspired by Tcl's array syntax. Arrays allow you to store values indexed by arbitrary keys—either numbers or strings.

12.1 Basic Syntax

Assignment:

```
SET arrayName(index) value
```

Access:

```
$arrayName(index)
```

Simple Array

```
SET fruits(1) "apple"
SET fruits(2) "orange"
SET fruits(3) "banana"

PUTS $fruits(1)    # apple
PUTS $fruits(2)    # orange
PUTS $fruits(3)    # banana
```

Output:

```
apple
orange
banana
```

12.2 Numeric Indices

Arrays can use numbers as indices, similar to traditional arrays:

Numeric Indices with Loop

```
SET colors(1) "red"
SET colors(2) "green"
SET colors(3) "blue"

SET i 1
WHILE $i <= 3 DO
  PUTS "colors($i) = $colors($i)"
  INCR i
END
```

Output:

```
colors(1) = red
colors(2) = green
colors(3) = blue
```

12.3 String Indices (Associative)

The real power of BCL arrays comes from using string keys:

Phone Directory

```
SET phone(John) "555-1234"
SET phone(Mary) "555-5678"
SET phone(Peter) "555-9012"

SET contact "Mary"
PUTS "Phone for $contact: $phone($contact)"
```

Output:

```
Phone for Mary: 555-5678
```

12.4 Variable Indices

You can use variables as array indices:

Variable as Index

```
SET data(Monday) "10"
SET data(Tuesday) "15"
SET data(Wednesday) "12"

SET day "Tuesday"
PUTS "Data for $day: $data($day)"
```

Output:

```
Data for Tuesday: 15
```

12.5 Expression Indices

Indices can be expressions, allowing for calculated array access:

Expression as Index

```
SET table(1) "A"
SET table(2) "B"
SET table(3) "C"

SET i 1
SET j [EXPR $i + 1]
PUTS $table($j)     # B

SET k [EXPR $i * 3]
PUTS $table($k)     # C
```

Output:

```
B
C
```

12.6 Multidimensional Arrays (Simulated)

BCL simulates multidimensional arrays using composite indices:

2D Matrix

```
SET matrix(1,1) "A"
SET matrix(1,2) "B"
SET matrix(2,1) "C"
SET matrix(2,2) "D"

PUTS "$matrix(1,1) $matrix(1,2)"
PUTS "$matrix(2,1) $matrix(2,2)"
```

Output:

```
A B
C D
```

12.7 Checking Element Existence

Use `INFO EXISTS` to check if an array element exists:

Check Existence

```
SET config(debug) "true"

IF [INFO EXISTS config(debug)] THEN
  PUTS "Debug is: $config(debug)"
ELSE
  PUTS "Debug not set"
END

IF [INFO EXISTS config(timeout)] THEN
```

```
  PUTS "Timeout is: $config(timeout)"
ELSE
  PUTS "Timeout not configured"
END
```

Output:

```
Debug is: true
Timeout not configured
```

12.8 Practical Examples

12.8.1 Configuration Settings

Application Config

```
SET config(host) "localhost"
SET config(port) "8080"
SET config(timeout) "30"
SET config(debug) "false"

PUTS "Server: $config(host):$config(port)"
PUTS "Timeout: $config(timeout)s"
PUTS "Debug mode: $config(debug)"
```

Output:

```
Server: localhost:8080
Timeout: 30s
Debug mode: false
```

12.8.2 Event Counter

Event Tracking

```
SET counter(login) "0"
SET counter(logout) "0"
SET counter(error) "0"

# Simulate events
SET counter(login) [EXPR $counter(login) + 1]
SET counter(login) [EXPR $counter(login) + 1]
SET counter(logout) [EXPR $counter(logout) + 1]
SET counter(error) [EXPR $counter(error) + 1]
SET counter(login) [EXPR $counter(login) + 1]

PUTS "Login: $counter(login)"
PUTS "Logout: $counter(logout)"
PUTS "Error: $counter(error)"
```

Output:

```
Login: 3
```

```
Logout: 1
Error: 1
```

12.8.3 Multiplication Table

Times Table

```
SET num 5
SET i 1
WHILE $i <= 10 DO
  SET table($num,$i) [EXPR $num * $i]
  PUTS "$num x $i = $table($num,$i)"
  INCR i
END
```

Output:

```
5 x 1 = 5
5 x 2 = 10
...
5 x 10 = 50
```

12.9 Arrays vs Lists

Tip

When to use Arrays:

- Key-value mappings (phone directory, configuration)
- Arbitrary indices (non-consecutive numbers or strings)
- Fast lookup by key

When to use Lists:

- Ordered collections
- Sequential processing
- Operations like sorting, joining, splitting

12.10 Arrays in Procedures

Arrays can be used with the `GLOBAL` keyword in procedures:

Global Arrays in Procedures

```
PROC show_person DO
  GLOBAL person
  PUTS "Name: $person(name)"
  PUTS "Age: $person(age)"
END

SET person(name) "Alice"
SET person(age) "30"
show_person
```

Output:

```
Name: Alice
Age: 30
```

Note

Array elements are stored as individual variables with names like `arrayname(index)`. They behave like regular variables and follow the same scoping rules.

12.11 The ARRAY Command

BCL provides the **`ARRAY`** command for advanced array manipulation, inspired by Tcl. This command offers powerful operations for working with arrays as complete structures.

12.11.1 ARRAY EXISTS

Check if an array exists (has at least one element):

```
ARRAY EXISTS arrayName
```

Returns `"1"` if the array exists, `"0"` otherwise.

Checking Array Existence

```
SET result [ARRAY EXISTS config]     # "0"
SET config(debug) "true"
SET result [ARRAY EXISTS config]     # "1"
```

12.11.2 ARRAY SIZE

Get the number of elements in an array:

```
ARRAY SIZE arrayName
```

Array Size

```
SET colors(red) "#FF0000"
SET colors(green) "#00FF00"
SET colors(blue) "#0000FF"

SET size [ARRAY SIZE colors]        # "3"
PUTS "Array has $size elements"
```

Output:

```
Array has 3 elements
```

12.11.3 ARRAY NAMES

Get a list of array indices, optionally filtered by a pattern:

```
ARRAY NAMES arrayName ?pattern?
```

Patterns support glob wildcards: `*` (any characters), `?` (one character), `[abc]` (character set).

Listing Array Indices

```
SET data(name) "John"
SET data(age) "30"
SET data(grade1) "8"
SET data(grade2) "9"

SET all [ARRAY NAMES data]             # All indices
SET grades [ARRAY NAMES data "grade*"] # Only grades
SET with_a [ARRAY NAMES data "*a*"]    # Containing 'a'

PUTS "All: $all"
PUTS "Grades: $grades"
PUTS "With 'a': $with_a"
```

Output:

```
All: name age grade1 grade2
Grades: grade1 grade2
With 'a': name age grade1 grade2
```

12.11.4 ARRAY GET

Get array contents as a list of index-value pairs:

```
ARRAY GET arrayName ?pattern?
```

Returns alternating indices and values, optionally filtered by pattern.

Getting Array Contents

```
SET colors(red) "255,0,0"
SET colors(green) "0,255,0"
SET colors(blue) "0,0,255"

SET all [ARRAY GET colors]
PUTS "All colors: $all"
```

Output:

```
All colors: red 255,0,0 green 0,255,0 blue 0,0,255
```

12.11.5 ARRAY SET

Populate an array from a list of index-value pairs:

```
ARRAY SET arrayName list
```

The list must have an even number of elements.

Setting Array from List

```
# Create array from list
ARRAY SET config "host localhost port 8080 debug true"

PUTS "Host: $config(host)"
PUTS "Port: $config(port)"
PUTS "Debug: $config(debug)"
```

Output:

```
Host: localhost
Port: 8080
Debug: true
```

Copying Arrays:

Copying an Array

```
SET original(a) "1"
SET original(b) "2"
SET original(c) "3"

# Copy array
SET data [ARRAY GET original]
ARRAY SET copy $data

PUTS "Copy size: [ARRAY SIZE copy]"
PUTS "copy(a) = $copy(a)"
```

Output:

```
Copy size: 3
copy(a) = 1
```

12.11.6 ARRAY UNSET

Delete array elements matching a pattern:

```
ARRAY UNSET arrayName ?pattern?
```

If no pattern is specified, deletes the entire array.

Selective Deletion

```
SET cache(temp_1) "data1"
SET cache(temp_2) "data2"
SET cache(perm_1) "data3"
```

```

PUTS "Before: [ARRAY SIZE cache]"     # "3"
ARRAY UNSET cache "temp_*"
PUTS "After: [ARRAY SIZE cache]"      # "1"
```

Output:

```
Before: 3
After: 1
```

Deleting Entire Array

```
SET myarray(1) "value1"
SET myarray(2) "value2"

ARRAY UNSET myarray

SET exists [ARRAY EXISTS myarray]     # "0"
PUTS "Array exists: $exists"
```

Output:

```
Array exists: 0
```

12.11.7 Practical Example: Configuration Manager

Configuration Manager

```
# Load configuration
ARRAY SET config "
    db_host localhost
    db_port 3306
    db_name myapp
    cache_enabled true
    cache_ttl 300
"

# Show all database settings
PUTS "Database Configuration:"
SET db_keys [ARRAY NAMES config "db_*"]
SET i 0
WHILE $i < [LLENGTH $db_keys] DO
    SET key [LINDEX $db_keys $i]
    PUTS "  $key = $config($key)"
    INCR i
END

# Show cache settings
PUTS ""
PUTS "Cache Configuration:"
SET cache_keys [ARRAY NAMES config "cache_*"]
SET i 0
WHILE $i < [LLENGTH $cache_keys] DO
```

```
    SET key [LINDEX $cache_keys $i]
    PUTS "  $key = $config($key)"
    INCR i
END
```

Output:

```
Database Configuration:
  db_host = localhost
  db_port = 3306
  db_name = myapp

Cache Configuration:
  cache_enabled = true
  cache_ttl = 300
```

Chapter 13

Binary Data

The **BINARY** command provides facilities for manipulating binary data in BCL, allowing conversion between binary representations and BCL values. This is essential for working with file formats, network protocols, and low-level data structures.

13.1 Overview

Syntax:

```
BINARY subcommand arguments
```

Two main subcommands:

- **FORMAT** - Build binary strings from BCL values
- **SCAN** - Extract BCL values from binary strings

13.2 BINARY FORMAT

Constructs a binary string according to a format specification.

Syntax:

```
BINARY FORMAT formatString ?arg arg ...?
```

The format string contains one or more field specifiers, each consisting of a type character and an optional count.

13.2.1 Format Codes

13.2.2 Basic Examples

String with Padding

```
# Null-padded string (10 bytes)
SET data [BINARY FORMAT a10 "hello"]
PUTS "Length: [STRING LENGTH $data]"

# Space-padded string (10 bytes)
SET data [BINARY FORMAT A10 "world"]
PUTS "Length: [STRING LENGTH $data]"
```

Code	Description
a	ASCII string, null padding
A	ASCII string, space padding
c	8-bit unsigned integers
s	16-bit integers, little-endian
S	16-bit integers, big-endian
i	32-bit integers, little-endian
I	32-bit integers, big-endian
H	Hex digits, high nibble first
h	Hex digits, low nibble first
x	Insert null bytes
X	Backspace (move cursor back)
@	Absolute position

Output:

```
Length: 5
Length: 10
```

8-bit Integers

```
# Pack bytes: 65='A', 66='B', 67='C'
SET data [BINARY FORMAT c3 "65 66 67"]
PUTS "Data: $data"
PUTS "Length: [STRING LENGTH $data]"
```

Output:

```
Data: ABC
Length: 3
```

Hexadecimal

```
# Convert hex string to binary
SET data [BINARY FORMAT H8 "deadbeef"]
PUTS "Length: [STRING LENGTH $data]"

# Use * to consume entire string
SET data [BINARY FORMAT H* "0123456789abcdef"]
PUTS "Length: [STRING LENGTH $data]"
```

Output:

```
Length: 4
Length: 1
```

13.3 BINARY SCAN

Extracts fields from a binary string and stores them in variables.

Syntax:

```
BINARY SCAN string formatString ?varName varName ...?
```

Returns the number of successful conversions.

13.3.1 Scan Examples

Extracting Strings

```
# Create binary data
SET data [BINARY FORMAT a10 "hello"]

# Extract 5 bytes
SET count [BINARY SCAN $data a5 var1]
PUTS "Extracted: '$var1'"
PUTS "Conversions: $count"

# Extract with trimming
SET data [BINARY FORMAT A10 "hello"]
SET count [BINARY SCAN $data A* var2]
PUTS "Trimmed: '$var2'"
```

Output:

```
Extracted: 'hello'
Conversions: 1
Trimmed: 'hello'
```

Extracting Integers

```
# Pack integers
SET data [BINARY FORMAT c5 "10 20 30 40 50"]

# Unpack integers
SET count [BINARY SCAN $data c5 numbers]
PUTS "Numbers: $numbers"
PUTS "Conversions: $count"
```

Output:

```
Numbers: 10 20 30 40 50
Conversions: 1
```

Extracting Hexadecimal

```
# Create binary data from hex
SET data [BINARY FORMAT H8 "cafebabe"]

# Extract as hex string
SET count [BINARY SCAN $data H* hex]
PUTS "Hex: $hex"
```

Output:

```
Hex: cafebabe
```

13.4 Practical Examples

13.4.1 Round-Trip Conversion

Pack and Unpack

```
# Original data
SET orig_a "10"
SET orig_b "20"
SET orig_c "30"

# Pack
SET packed [BINARY FORMAT c3 "$orig_a $orig_b $orig_c"]
PUTS "Packed length: [STRING LENGTH $packed]"

# Unpack
SET count [BINARY SCAN $packed c3 unpacked]
PUTS "Unpacked: $unpacked"
PUTS "Conversions: $count"
```

Output:

```
Packed length: 3
Unpacked: 10 20 30
Conversions: 1
```

13.4.2 Data Structure Serialization

Simple Record Format

```
# Create record: name(10 bytes) + age(8-bit) + id(8-bit)
SET name "Alice"
SET age "25"
SET id "42"

SET record [BINARY FORMAT a10c2 "Alice" "$age $id"]
PUTS "Record size: [STRING LENGTH $record]"

# Read record
SET count [BINARY SCAN $record a10c2 stored_name data]
SET stored_age [LINDEX $data 0]
SET stored_id [LINDEX $data 1]

PUTS "Name: '$stored_name'"
PUTS "Age: $stored_age"
PUTS "ID: $stored_id"
```

Output:

```
Record size: 12
Name: 'Alice'
Age: 25
ID: 42
```

13.4.3 Hex Dump Utility

Hex Dump

```
PROC hexdump WITH data DO
    SET len [STRING LENGTH $data]
    SET i 0

    WHILE $i < $len DO
        # Get one byte
        SET byte [STRING INDEX $data $i]
        SET tmp [BINARY FORMAT a1 $byte]
        BINARY SCAN $tmp H2 hex

        PUTSN "$hex "
        INCR i

        # New line every 16 bytes
        IF [EXPR $i % 16] == 0 THEN
            PUTS ""
        END
    END
    PUTS ""
END

SET data "Hello World!"
hexdump $data
```

Output:

```
48 65 6c 6c 6f 20 57 6f 72 6c 64 21
```

13.5 Endianness

BCL supports both little-endian and big-endian byte ordering:

Tip

Little-Endian (s, i):

- Least significant byte first
- Common on x86/x64 architectures
- Example: 0x1234 stored as [0x34, 0x12]

Big-Endian (S, I):

- Most significant byte first
- Common in network protocols (network byte order)
- Example: 0x1234 stored as [0x12, 0x34]

For cross-platform data exchange, prefer big-endian (S, I).

13.6 Limitations

Note

Important: BCL uses null-terminated C strings internally. This means:

- Binary data containing null bytes (0x00) in the middle may be truncated
- This mainly affects 16/32-bit integers with small values
- **Workaround:** Use 8-bit integers (c) or hexadecimal (H) when possible

The current implementation does NOT support:

- Float/double types ('f', 'd')
- Binary digit representation ('b', 'B')

13.7 Common Patterns

13.7.1 Checksum Calculation

Simple Byte Sum

```
PROC checksum WITH data DO
    SET sum 0
    SET i 0

    WHILE $i < [STRING LENGTH $data] DO
        SET byte [STRING INDEX $data $i]
        BINARY SCAN $byte c val
        SET sum [EXPR $sum + $val]
        INCR i
    END

    RETURN [EXPR $sum % 256]
END

SET msg "Hello"
SET cs [checksum $msg]
PUTS "Checksum of '$msg': $cs"
```

Output:

```
Checksum of 'Hello': 244
```

13.7.2 Simple Protocol Header

Message Protocol

```
# Create message: type(8-bit) + length(8-bit) + data
SET msg_type "1"
SET msg_data "Hello"
SET msg_len [STRING LENGTH $msg_data]

SET packet [BINARY FORMAT c2a* "$msg_type $msg_len" $msg_data]
PUTS "Packet size: [STRING LENGTH $packet]"

# Parse message
BINARY SCAN $packet c2a* header payload
SET type [LINDEX $header 0]
SET len [LINDEX $header 1]

PUTS "Type: $type"
PUTS "Length: $len"
PUTS "Payload: $payload"
```

Output:

```
Packet size: 7
Type: 1
Length: 5
Payload: Hello
```

13.8 Quick Reference

Type	Size	Purpose
a/A	variable	ASCII strings
c	1 byte	Unsigned 8-bit integers
s/S	2 bytes	16-bit integers (little/big)
i/I	4 bytes	32-bit integers (little/big)
H/h	variable	Hexadecimal digits
x	variable	Padding/spacing
X	-	Backspace cursor
@	-	Absolute positioning

Count Modifiers:

- Number: exact count
- * : consume all available
- Omit: default to 1

Appendix A

Installation

A.1 System Requirements

BCL can run on:

- PC platforms (Windows, Linux, macOS)
- Embedded systems with sufficient resources
- Microcontrollers (resource-constrained version)

A.2 Installation Steps

A.2.1 From Binary

1. Download the BCL distribution for your platform
2. Extract the archive
3. Add the `bin` directory to your PATH
4. Verify installation: `bcl -version`

A.2.2 From Source

Consult the BUILD.md file in the source distribution.

Appendix B

Troubleshooting

B.1 Common Errors

Variable not found Check spelling and ensure the variable has been set with `SET`

File not found Verify the file path and permissions

Syntax error Check for missing `END` keywords and proper command syntax

Type mismatch Ensure numeric operations use valid numbers

Appendix C

Differences from Tcl

BCL is inspired by Tcl but has several key differences:

- **Syntax**: Uses BASIC-style keywords (`IF...THEN...END` instead of braces)
- **Case-insensitivity**: Commands are not case-sensitive
- **Variable expansion**: Only `$var` form (not `${var}`)
- **Procedure invocation**: Direct name invocation (no `CALL`)
- **Block ending**: All blocks end with `END`
- **Optional parameters**: Use `@param` prefix

Appendix D

Future Enhancements

Planned features for future BCL versions:

- Dictionary/associative array data type
- Namespace support
- Object-oriented programming features
- More comprehensive standard library
- Debugging and profiling tools
- Package management system